AD-A219 929

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 8, 1990	Final Report, 15 Sep 89 to 14 Dec 89

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
INTELLIGENT, REAL-TIME PROBLEM SOLVING	F49620-89-C-0121 62702F 5581/00

6. AUTHOR(S)

Paul Cohen
David M. Hart

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Massachusetts
Computer and Information Sciences Dept.
Amherst, MA 01003

8. PERFORMING ORGANIZATION REPORT NUMBER

AFOSR·TR· 90-0324

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
Building 410
Bolling AFB, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

F49620-89-C-0121

DTIC
ELECTE
MAR 29 1990
S
D
D

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A testbed for intelligent, real-time problem solving systems has been enhanced for use by the broader research community. The testbed, part of the Phoenix system, simulates forest fires and autonomous agents who try to control them. Under this contract, the testbed has been modularized for portability to other researchers using Explorers or MicroExplorers[1]. Instrumentation has been added for experimentation and baseline scenarios developed for typical real-time problems found in this domain. A reference manual for the testbed has been written.

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Intelligent, Real-time Problem Solving

Paul Cohen, *Principle Investigator*
David M. Hart, *Lab Manager*

Experimental Knowledge Systems Laboratory
Computer and Information Science Dept.
Univ. of Massachusetts
Amherst, MA 01003

March 8, 1990

Accesion For		
NTIS CRA&I		☑
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

Abstract

A testbed for intelligent, real-time problem solving systems has been enhanced for use by the broader research community. The testbed, part of the Phoenix system, simulates forest fires and autonomous agents who try to control them. Under this contract, the testbed has been modularized for portability to other researchers using Explorers or MicroExplorers[1]. Instrumentation has been added for experimentation and baseline scenarios developed for typical real-time problems found in this domain. A reference manual for the testbed has been written.

---

[1] Explorer and MicroExplorer are trademarks of Texas Instruments Corporation

1.0 Statement of Work and Status of Research Effort

Four tasks comprised our Phase I effort to make the Phoenix environment accessible to all IRTPS participants. Our fulfillment of these tasks is summarized by section below. A more complete description of our research effort can be found in Appendix A, which is our report to the IRTPS workshop in Santa Cruz, and Appendix B, a reference manual for the Phoenix testbed.

1.1 Make the Phoenix System Portable

We have made Phoenix portable to any TI Explorer (color or B&W) and to the MicroExplorer. It is packaged into one system that includes all non-standard (non-proprietary) support code we use to run it. The complete system can be provided to other laboratories and research groups on tape as a TI Load Band. We have provided supporting documentation for the testbed (see Appendix B), enhanced the on-line help facilities, and annotated the code.

1.2 Instrument the System

Phoenix is now instrumented at the three levels discussed in our contract proposal. Using tools provided by the Explorer, we meter the system performance (implementation level). We measure performance of problem solving in the domain by assessing such factors as the amount of forest burned and resources consumed by agents (time, fuel, etc.). At the solution level, we provide measurement tools that are suited to our own problem solver and hooks for the solutions developed by other researchers. The instrumentation is built into a m user interface that allows enabling and disabling of measurement at each vel independently, providing flexible control over the duration and level. For more information, see Section 3.3 of Appendix A.

1.3 Provide Baselines

We have developed baseline scenarios for several situations that are both characteristic of this domain and require timely response to environmental changes. One of these scenarios, described in detail in Section 3.4 of Appendix A, involves a single fire whose profile changes dramatically due to shifting environmental conditions, so that the nature of the threat changes from the potential loss of forest to loss of populated areas. A problem solver fighting this fire must respond in a timely way to its unpredictable change to avoid losing a highly-valued area. Another scenario we have developed presents the problem solver with multiple fires and limited fire-fighting resources which must be managed efficiently to prevent one or more of the fires from spreading out of control. By limiting the number of fire-fighting agents available, and limiting the amount of fuel each can carry (requiring them to refuel periodically), this

scenario forces the problem solver to allocate its resources wisely in order to control the fires.

A scripting capability allows the user to create and store such scenarios for establishing new baselines, developing new real-time problem solving solutions, and testing the effectiveness of those solutions. Scripts give control over environmental factors such as when and where fires start and wind characteristics, and the resources available for fire-fighting (how many agents of each type, what are their speeds, fuel capacities, fields of view). The timing of environmental changes is specified in scripts, allowing the user to control when events occur in the simulation for testing purposes. Instrumentation functions can be run within scripts to gather data useful for development and testing.

1.4 Modularize System Components

We have modularized Phoenix so that other researchers can work with all or part of it. The five levels of the system are described in Section 1 of Appendix A. Code for each of these levels has been separated into self-contained building blocks. The first two levels are the fire simulation testbed, and are comprised of the system kernal (graphic user interface), the task scheduler, maps of the environment, and the fire simulation task. Researchers interested in designing and implementing agent architectures for real-time problem solving could test them in this simulated, instrumented testbed environment (see Appendix B). The next level is a generic agent architecture shell - a set of functional components common to all agents. Code to interface these components with the testbed is included with this level, so that researchers interested in working with our functional decomposition of agent capabilities need only instantiate these components -- sensors, effectors, reflexes, and cognitive capabilities -- with their own versions. The fourth level includes our versions of these components. This provides a specific agent architecture that is distinctive primarily for the planning style used in the cognitive component (skeletal planning with delayed commitment to specific actions[2]). Researchers interested in using our planning style as well as agent architecture could work with the first four levels, creating their own agent types and organizing them according to their research interests. The fifth level is the organization of fire fighting agents; we use a hierarchical organization in which a single fireboss agent with a global view directs the activities of semi-autonomous field agents with local views. Researchers interested in working with our solution to real-time problem solving in this domain would use all five levels to replicate and/or extend our work. Appendix A of "The Phoenix Testbed" (Appendix B of this

---

[2] For more on the our cognitive architecture, see "Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments", Cohen, et al., *AI Magazine*, Vol. 10, No. 3, pgs. 32-48.

document) shows the division of functionality into different code modules that can be used to build the various levels of the system described above.

2.0 Workshop and Technical Reports

Paul R. Cohen, A.E. Howe, and David M. Hart. Intelligent Real-Time Problem Solving: Issues and Examples. *Intelligent Real-Time Problem Solving: Workshop Report,* edited by Lee D. Erman, Santa Cruz, CA, November 8-9, 1989, pages IX-1 -- IX-34.

Michael Greenberg and David L. Westbrook. The Phoenix Testbed. Technical Report #90-19, Experimental Knowledge Systems Laboratory, Dept. of Computer and Information Science, Univ. of Massachusetts.

These reports are included as appendices in this document.

3.0 List of Professional Personnel

Paul R. Cohen, principle investigator, co-authored the workshop report on the Phoenix system (Appendix A) and presented it at the Santa Cruz workshop.

David M. Hart, lab manager, co-authored the workshop report and supervised the research activities reported here.

Adele E. Howe, graduate research assistant, co-authored the workshop report and participated in the Santa Cruz workshop.

Michael Greenberg, staff programmer, worked extensively to modularize the system and make it portable to other research groups, and co-authored the testbed documentation (Appendix B).

David L. Westbrook, staff programmer, developed instrumentation and facilities to support baseline scenarios and experimentation and co-authored the testbed documentation.

4.0 Interactions

Paul Cohen presented "Intelligent Real-Time Problem Solving: Issues and Examples" at the Intelligent Real-Time Problem Solving Workshop in Santa Cruz, November 8-9, 1989.

Gerald M. Powell is a visiting faculty member who is here under the Secretary of the Army Research and Study Fellowship Program. Dr. Powell, who works

for the Center for Command, Control, and Communications Systems, CECOM, Ft. Monmouth, New Jersey, has been investigating computational approaches to various problems in battlefield planning for the past five years, and is very interested in the present capabilities and further design and development of Phoenix.

We had five meetings in September and October, 1989, with members of Victor Lesser's research group at Umass (Robert Whitehair and Keith Decker) who are exploring techniques for sophisticated real-time control in the Distributed Vehicle Monitoring Testbed, and are interested in experimenting with them in the Phoenix testbed.

Intelligent Real-time Problem Solving:
Issues and Examples[1]

Paul R. Cohen, Adele E. Howe, David M. Hart
Experimental Knowledge Systems Laboratory
Department of Computer and Information Science
University of Massachusetts, Amherst

cohen@cs.umass.edu
413 545 3638

1. Introduction

This report presents our work on real time problem solving (IRTPS). The topic is fundamentally challenging in the sense that it probably cannot be completely addressed within the established knowledge-based and logicist paradigms, but w'' require methodological, theoretical, and technical developments. Accordingly, this report looks at IRTPS from all these perspectives. Because readers will have different interests, each section of the report is independent of all sections except this Introduction. Section 2 offers a definition of IRTPS. Section 3 describes our real-time testbed, including its current status and portability, and our timetable for making it generally available. Section 4 discusses the architecture we have developed for real-time agents and our near-term research goals. Section 5 is devoted to methodological issues, specifically, how characteristics of environments constrain the design of agents (including an assessment of the pros and cons of simulated environments), how to evaluate IRTPS systems, and the need for analytic models of agent architectures.

The task environment for much of our research is a simulation of forest fires. The task is to control simulated fires by deploying simulated agents, including "smart" bulldozers, fuel carriers, and airplanes. (Smart agents have the simulated physical abilities of, say, bulldozers, and some of the simulated mental abilities of their human operators.) This is a realtime problem in the basic sense that *the environment changes while agents think and act.* If agents think too long, the fires get too big to control. If they don't think long enough, their plans may be flawed and their actions may be less effective. (Section 2 refines this basic definition of the real-time problem.)

The Phoenix system comprises five levels of software:

> DES -- the discrete event simulator kernel. This handles the low-level scheduling of agent and environment processes. Agent processes include sensors, effectors, reflexes, and a variety of cognitive actions. Environment processes include fire, wind, and weather. The DES provides an illusion of simultaneity for multiple agents and multiple fires.

> Map -- this level contains the data structures that represent the current state of the world as perceived by agents, as well as "the world as it really is." Color graphics representations of the world are generated from these data structures.

> Basic agent architecture -- a "skeleton" architecture from which agents, such as bulldozers, airplanes, and firebosses are created. The agent

architecture provides for sensors, effectors, reflexes, and a variety of styles of planning.

Phoenix agents -- the agents we have designed (and are designing) for our own RTPS experiments.

Phoenix organization -- currently we have a hierarchical organization of Phoenix agents, in which one fireboss directs (but does not control) multiple agents such as bulldozers. Each Phoenix agent is autonomous and interprets the fireboss's directions in its local context, while the fireboss maintains a global view. A related project is looking at multiple firebosses and distributed control.

The Phoenix environment (the DES and map level), the basic agent architecture, and Phoenix agents are independent software packages that we offer to other researchers (see Section 3.3). We will offer instrumentation for these components of the Phoenix system by the end of Phase I of the IRTPS initiative.

Our research on IRTPS is part of a larger project whose goal is to develop a sound basis for the design of AI agents. We are analyzing agents in terms of the *behavioral ecology view* shown in Figure 1. This view encourages us to ask how the characteristics of environments (including time) constrain the design and behavior of agents. (We compare this view with the S/E model of Rosenschein, Hayes-Roth and Erman, in Section 5). When we speak of a sound basis for design, we mean the ability to predict how modifying the architecture of an agent will change its behavior in a given environment. Currently, we do this by building models that relate the architecture of an agent to behaviors. The methodological implications of the behavioral ecology view and of modelling are discussed in Section 5.

Figure 1. The three components of an agent's behavioral ecology.
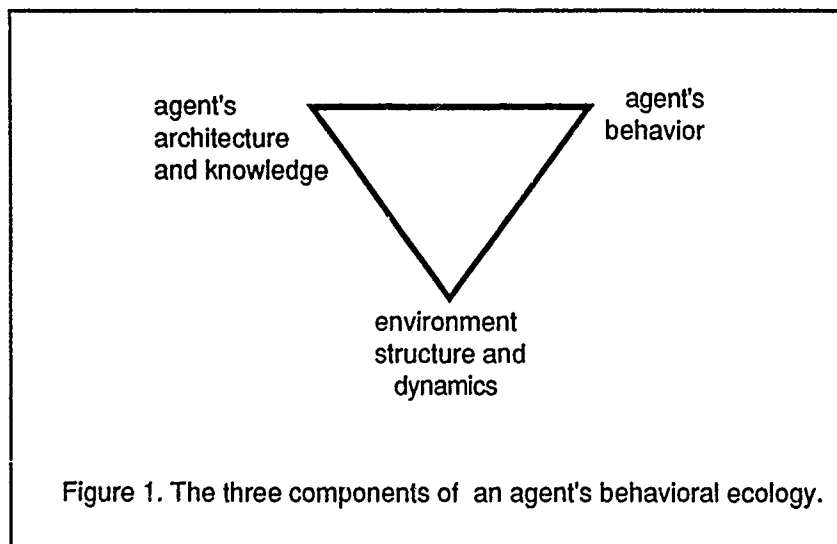
2. Definitions.

We begin this section with definitions of real-time problem solving and "the real time problem." Next we examine some terms that are common in the IRTPS literature, such as deadline, predictability, and time scale. These terms are vague, and it is often difficult to tell whether they are intended as descriptions of an agent's environment or its behavior. We propose six classes of terms that should, we hope, reduce the vagueness and ambiguity of previous discussions of IRTPS. Lastly, we show how the behavioral ecology view helps us compare and organize different approaches to IRTPS.

2.1 IRTPS and the Real-time Problem

A crude definition of IRTPS was mentioned in the introduction:

The environment changes while agents think and act.

But this doesn't adequately convey the impact of changes in the environment upon the agents. For example, while a bulldozer thinks about how to avoid a fire, the position of the fire changes, and in some cases the bulldozer can be overrun. A better definition makes explicit the value of problem solving and how it is affected by changes in the environment:

> *The value of problem solving is a function of what the problem solver does --- its thinking and acting --- and one or more parameters in the environment, at least one of which changes during problem solving.*

Note that this definition makes no direct reference to time. This is because time is itself an indirect way of talking about changes in the environment during problem solving, and it is these changes, not the passage of time, that affect the value of problem solving. When we say, "The fire is currently consuming 10 acres per hour," we do *not* mean that an hour is worth ten acres. Time itself has no inherent value. Time provides us a scale on which to measure events that do have value. Consider an analogy to distance. We might say, "As we drive down this street, property values increase by $10,000 a block," but we would not say that a block (e.g., 200 yards) is worth $10,000. Throughout this document we will try to avoid giving the impression that time has any value. We will try to foster the view that time (like distance) is just a scale on which to plot changes in value.

Of course, we can define value to be a function of time, as in real-time operating systems, but typically we measure value in terms of money, acreage, real estate, lives saved or lost, and so on. Unlike time, these value functions may be nonlinear, even discontinuous. This leads to the following definition of the "real time problem":

> *The value of problem solving does not always increase, nor does it always decrease, during problem solving; thus simple strategies such as "work for as long as possible," or "solve the problem as quickly as possible," will generally not maximize value.*

Example: Measuring value in terms of the area of burned forest, we might collect statistics on the relationship between area burned and the amount of time that elapses while the fireboss selects a plan. (For simplicity, we will plot elapsed time against area saved instead of the area burned, so value increases on the y axis:)
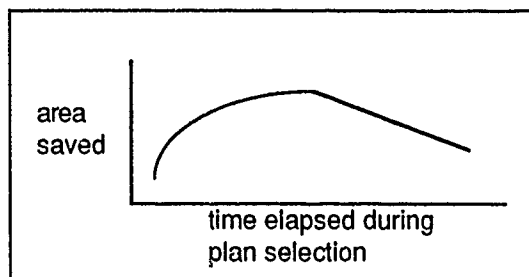
Apparently, a little time devoted to plan selection is worthwhile. After a point, however, thinking longer doesn't save more forest. While this "inverted U" could have many causes, the important point is that it seems to be characteristic of real time tasks.

The inverted U function seems more representative of soft deadlines than hard ones. Value decreases slowly when a soft deadline is missed, precipitously when a hard deadline is missed. In Section 2.2 we formalize and illustrate a hard deadline in the Phoenix environment.

It follows from this definition of the real time problem that an agent must have control of the amount of time it devotes to problem solving. In terms of the previous example, an agent should spend "just enough" time on plan selection---the amount of time that corresponds to the highest point on the curve. Although this picture is an oversimplification (we will discuss some of the complexities later) it does illustrate that if an agent cannot control the amount of time it spends on problem solving, it cannot affect the value of its problem solving.

What does it mean for an agent to control the amount of time it spends on problem solving. First, it does *not* mean that the agent controls the rate at which time passes. We assume this is beyond every agent's control. Instead we mean that an agent can with "one eye on the clock" decide whether to run a process or continue running an interrupted process.

Example: The Phoenix agent architecture provides for multiple *execution methods* to achieve any goal. For example, Phoenix has several path planning methods. Execution methods for a given task require different amounts of time. More precisely, they differ in the amounts of time that are expected to elapse before each terminates. One way that Phoenix agents control the amount of time they spend on problem solving is to base the selection of execution methods on their estimated time requirements.

Estimates figure heavily in this example, and in IRTPS in general. What remains to be seen, in the course of this research, is what characteristics of the environment affect the quality of estimates, and what characteristics of IRTPS architectures affect their dependence on the quality of estimates.

2.2 Describing the Environment and Agents

Although our approach is to design agents for specific environments, we have been content to describe environments at two levels of abstraction, both inadequate for design. We believe this is common in the IRTPS literature.

The levels, with examples, are:

Implementation-specific: When a cell ignites, the simulator figures out when its knights-tour-neighbors are going to ignite. It calculates the rate of spread of the newly-ignited cell to its neighbors, accounting for weather, slope, fuel type, etc.

Apple-pie general: The Phoenix environment is characterized by unpredictable events, real-time constraints, and hard and soft deadlines.

Neither level of description of the environment is appropriate for design. From the first kind of description you can model the structure and dynamics of the environment, so it is genuinely useful. But the second kind of description is actually misleading without a lot of clarification, as the following examples show.

At the IRTPS Workshop[2], the Working Group on Architectures made a list of characteristics of environments:

lots of data

low signal to noise ratio

unpredictable rates at which data arrive (varying quantity of data)

hard and soft deadlines

time-dependent value

spectrum of predictability

---

[2]Pasatiempo, Santa Cruz, California. November 6 & 7, 1989.

6

incompleteness in data

multiple time scales

combinatoric proliferation of things to attend to

Most of these seem self-explanatory. However, most could be interpreted as descriptions of the agent as well as descriptions of the environment. Take "multiple time scales." Our definition of time scale is the average time between causal event cycles that have value for the agent. Some cycles are very short (e.g., the time between moving into a fire and getting burned) and some are much longer (e.g., the time between a wind shift and the recognition of failure of a fire-fighting plan). But it doesn't make sense to talk about time scales independent of an agent; specifically, independent of the value of events to the agent. Without the concept of value, there's no way to classify the limitless number of causal event cycles, and so the distribution of time scales is uniform. The concept of value enables us to select classes of events—those that have value to the agent—and compute the average length of their causal event cycles. Scale depends on the agent design. It is not an inherent property of environments[3].

Which of the other characteristics listed above is inherent to environments, and which depend on the agent design? For some, both interpretations make sense. When we say "lots of data," we could mean two things: First unlike environments like the blocks world, a lot is happening in the Phoenix environment—there's a lot for the agent to attend to. This seems to be a description of an inherent characteristic of the environment. But we might also mean that in this environment, the agent's sensors can take in more stuff than it can process. So "lots of data" can be interpreted as a characteristic of the environment or as a potential problem for the agent.

Next, consider the "predictability" characteristic. When we say events are unpredictable, we are usually mean "unpredictable for this agent." But, again, we might mean "unpredictable for *any* agent." Once again, we must take care to separate the inherent characteristic of the environment—the one that would constrain any agent—from the potential problem that the environment poses the agent.

The resolution of this ambiguity should provide us with the appropriate level of description of environments for doing design.

There is a predicate called, U_e, that takes environmental events as arguments. $U_e(e)$ means that no agent can predict event e.

There is a predicate U_a that takes environmental events and agents as arguments. If $U_a(e,a)$ then agent a cannot predict event e.

---

[3]We are grateful to Les Gasser for pointing this out.

By definition, $U_e(e)$ implies $U_a(e,a)$ for all a. But it is not the case that $U_a(e,a)$ implies $U_e(e)$. This means, at the very least, that we have to be careful when we say an environment is characterized by unpredictability. More importantly, it points to a gap in our understanding of the agent-environment interaction: If $U_a(e,a)$ and not $U_e(e)$, there must be something about event e that makes it unpredictable to agent a, and if we want to design an agent a' for which $Ua(e,a')$ is false, we have to know why $Ua(e,a)$ is true. To designers, it helps to know $U_e(e)$, but knowing $U_a(e,a)$ doesn't help us fix the problem. We need to know something else. For example, if changes in the position of the fire are unpredictable to an agent, and we view this as a problem, then we need to know why the changes are unpredictable. Two contributing factors may be the limited field of view of agents and the statistical distribution of changes in wind speed and direction. One is an architectural characteristic, the other an environmental characteristic, and together they produce $U_a(\text{fire-position},\text{agent})$.

Terms like "unpredictable" are just shorthand for problems faced by particular agents, not characteristics of the environment, except when they are universally quantified over agents. From the standpoint of design, they don't help us much. Instead we will develop a vocabulary to describe environments in *problem-independent* terms. When we say that an event has a statistical distribution we do not imply anything about the architecture of an agent, or anything about the problem that may arise for an agent as a result of the event having a statistical distribution. We suggest six classes of terms:

> **Environment characteristics (ECs).** These are problem-independent, architecture independent descriptors of the environment. For example, a parameter (say, windspeed) changes aperiodically. A counterexample: Windspeed is unpredictable.

> **Architecture characteristics (ACs).** These are problem-independent, environment-independent descriptors of the architecture. For example, the architecture has a random access memory of limitless capacity; or, the plan selection mechanism is bounded in computation time. A counterexample: the error recovery mechanism exhibits graceful degradation. This is a counterexample because graceful degradation implies something about the problem you are trying to solve.

> **Problems.** A problem is a shorthand for an undesirable behavior, that is, an undesirable interaction between a particular agent and a particular environment. For example, unpredictability is a shorthand for interactions in which, because an agent did not anticipate an environmental event, some negative consequence occured.

> **Inherent problems.** An inherent problem is a problem that we believe all agents face, that is, an undesirable interaction between *any* agent and a particular environment.

> **Solutions.** A solution is a shorthand for a desirable behavior that we, as designers, want to see instead of some undesirable behavior—the

problem. For example, a fast sense-act loop is sometimes a solution to the unpredictability problem.

Solution realizations. A solution realization is one or more architecture characteristics or modifications to architecture characteristics; in short, what we intend to do to the architecture to ensure that the solution (which is a behavior, remember) will occur when we want it to.

In Section 5.1, we illustrate how design of a real-time mechanism proceeds from an informal description of a problem, through a formal description in terms of ECs, ACs and Problems, to solutions and solution realizations.

2.3 The Behavioral Ecology Triangle Organizes and Justifies IRTPS Approaches

We can characterize the dozens of approaches to IRTPS in terms of the behavioral ecology triangle in Figure 1. First, what behaviors do we want from IRTPS systems? Second, what characteristics of the environment make particular behaviors necessary or desirable? Third, what architectural decisions can designers make to achieve the desired behaviors in the given environment?

Example: A desired behavior is for Phoenix agents to meet their deadlines. Two characteristics of the Phoenix environment make this desirable: First, most fires must be contained by the coordinated efforts of several agents. Second, fires spread in such a way that if one agent is very late, the work of others is jeopardized. Another characteristic of the environment conspires against coordinated effort: unpredictable changes in parameters such as wind speed and direction differentially affect the progress of agents. The architectural decisions that allow Phoenix agents to meet deadlines despite unexpected events are discussed in Section 4.2.

The behavioral ecology view provides a framework for organizing the real time literature. For example, we can ask what characteristics of the environment make anytime or approximate processing behaviors necessary or desirable, and what architectural choices are needed to implement anytime or approximate behaviors in particular environments. But we have found that the principal advantage of the behavioral ecology view is that it forces us to justify our design decisions in terms of agents' environments.

Example: It is not uncommon to claim that IRTPS requires a behavior called "graceful degradation." (Other candidates are anytime or approximate behavior). Too often, the next step is to build an architecture that implements this behavior in some environment. This is backwards. The first step must always be to ask whether the environment makes graceful degradation (or anytime, or approximate behavior) necessary or desirable.

3. A Real Time Testbed.

This section describes describes the Phoenix testbed from several perspectives. Section 3.1 describes how the testbed appears to a user. Section 3.2 focuses on how the discrete event simulator manages simulation time and cpu time for multiple pseudo-parallel processes. Section 3.3 describes three levels of instrumentation for the testbed. Section 3.4 presents a "baseline scenario," that can be run again and again under different conditions to test real-time architectures. Section 3.5 addresses portability issues. The structure and implementation of the testbed is independent of the architecture of Phoenix agents; indeed, we hope that other researchers will use the testbed as an environment in which to test their own agent architectures. For this reason, we will postpone discussing the Phoenix basic agent architecture until Section 4.

3.1 The Appearance and Behavior of the Testbed.

If you watch the Phoenix system run, this is what you will see: A color representation of Yellowstone National Park, in which fires are spreading and several bulldozers, fuel-carriers, and other agents are travelling and cutting fireline. You will see different kinds of vegetation coded by color. You will also see roads and rivers of different sizes, elevation lines, lakes, houses, and watchtowers. Status windows present elapsed time, wind speed, and wind direction. You have full control over the resolution of your view; for example, you can see the entire map at low resolution or just a few acres at high resolution. You can see the environment as it really is, and as it is perceived by one or more of the agents (which have limited fields of view). Figure 2 shows a view of an area of the park, unfortunately not in color (but see [Cohen, 1989] for color pictures). The grey region at the bottom of the screen is the northern tip of Yellowstone Lake. The thick grey line that ends in the lake is the Yellowstone River. The Grand Loop Road follows the river to the lake, where it splits. The Smokey the Bear symbol in the bottom left corner marks the location of the fireboss, the agent that directs and coordinates all others. Two bulldozers are shown cutting fireline around a fire in this figure. Two other bulldozers are parked near the fireboss, along with a plane and a fuel carrier.

The Map level of the Phoenix environment, from which the graphics representations of the environment are generated, is constructed from Defense Mapping Agency data. Because it includes ground cover, elevation, moisture content, wind speed and direction, and natural boundaries, we have been able to construct a moderately realistic simulation of forest fires (but see Section 5 for a distinction between realism and accuracy). For example, real fires and our simulated fires spread more quickly in brush than in mature forest, are pushed in the direction of the wind and uphill, burn dry fuel more readily, and so on. These conditions also determine the probability that the fire will jump fireline and

r tural boundaries; and the intensity of the fire (which is coded by color in the simulation) The physical abilities of fire-fighting agents are also simulated accurately; for example, bulldozers move at a maximum speed of 40 kph in transit (on the back of a truck), 5 kph traveling cross-country, and 0.5 kph when cutting fireline.

Recently, we have implemented some realistic weather factors, specifically, lightning strikes which start fires with some frequency, and rain, which affects the moisture and thus the friability of fuels.

Fires are fought by removing one or more of the things that keep them burning: fuel, heat, and air. Cutting fireline removes fuel. Dropping water and flame retardant removes heat and air, respectively. In major forest fires, controlled backfires are set to burn areas in the path of wildfires and thus deny them fuel.

In the past, fire-fighting agents were inexhaustible, but recently we have started to model their consumption of resources. The following example of monitoring fuel levels and refueling conveys the flavor of problem solving within and among Phoenix agents.

Example: Bulldozers monitor their own fuel levels and notify the fireboss when their tank drops below a preset level. Upon reciept of the "I'm low on fuel" message, the fireboss marks the bulldozer with a status of "needs-refueling" and when the bulldozer becomes idle (i.e. completes the segment of fireline it is working on), the fireboss selects a refueling plan for that bulldozer. This involves allocating an available fuel-carrier, calculating a rendezvous point on a road near the bulldozer, telling the bulldozer where to go and who to look for, telling the fuel-carrier where to go, and waiting for an acknowledgement from the bulldozer that it has received fuel. The bulldozer and fuel-carrier then interact through the following process: the bulldozer notices the rendezvous, requests service, and waits for a service-complete acknowledgement from the fuel-carrier. The fuel-carrier arrives at the destination, waits for service requests, queues them up if necessary (not currently utilized), transfers fuel via a pump-effector, terminating when either the bulldozer is not present or leaves, the refueling tank goes dry, the bulldozer's tank is full, or the requested amount is pumped. The fuel-carrier then tells the bulldozer it has finished and the bulldozer in turn tells the fireboss that the refuleing task has been completed.

3.2 The Implementation of the Testbed

Underlying the Phoenix testbed is a discrete event simulator (DES) that creates the illusion of a continuous world, where natural processes and agents are acting in parallel, on serial hardware (currently a Texas Instruments Explorer II Color Lisp Machine, but see Section 3.5). In the simulation, fires burn continuously over time and agents act in concert to control it. Some of these actions are physical, as

in digging fireline and cutting trees. In parallel to these physical actions, agents perceive, move, react to perceived stimuli, and think about what action(s) to execute next.

The DES manages two types of time: cpu time and simulation time. CPU time refers to the length of time that processes run on a processor. Simulation time refers to the ``time of day" in the simulated environment. The illusion of continuous, parallel activity on a serial machine is maintained by segregating each process and agent activity into a separate task and executing them in small, discrete time quanta, ensuring that no task ever gets too far ahead or behind the others. The default setting of the synchronization quantum is five simulation-time minutes, so all tasks are kept synchronized to within five simulated minutes of one another.

The quantum can be increased, which improves the cpu utilization of tasks and makes the simulator run faster, but this increases the simulation-time disparity between tasks, magnifying coordination problems such as communication and knowing the exact state of the world at a particular time. Conversely, decreasing the quantum reduces how ``out of synch" processes can be, but increases the running time of the simulation.

Within the predefined time quantum, all simulated parallel processes begin or end at roughly the same simulation time. Types of tasks differ in how they are ``charged for" cpu time and simulation time. Sensory tasks run for very short intervals of simulation time, after which they are rescheduled; this gives them a high sampling rate compared to the rate at which the world is changing. Effector tasks may use very little simulation time, or the full synchronization quantum. Fire tasks always run for the full synchronization quantum.

All these tasks are allotted as much cpu time as they need by the DES; there is no constant proportionality between the simulation time and the cpu time they require. To see why, note that fires are implemented as cellular automata, so that the cpu time required to calculate the spread of the fire depends on the size of the fire. It may take only a fraction of a second of cpu time to calculate five simulation-time minutes of burning for a small fire, but several cpu seconds to calculate the five simulation-time minutes for several large fires. Similarly, the amount of cpu time required to calculate a few simulation-time seconds of sensor processing depends on the type of sensor being simulated, so there is no constant proportionality between simulated sensor time and cpu time.

In contrast, there is a constant proportionality between the cpu time allocated to cognitive tasks and simulation time. This is because we want to "charge" agents at a fixed rate for thinking. Because cognition and other processes, such as the fire, are simulated parallel processes, they are always allocated the same amount of

simulation time. So when both have run for their allocated times

$$\text{elapsed-simulation-time(cognition)} = \text{elapsed-simulation-time(fire)}$$

as measured by the simulation time clock. Although these processes could take arbitrary amounts of cpu time, it is advantageous to impose a strict relationship between cpu time and the simulation time of the planner and the fire. Thus,

$$\text{elapsed-simulation-time(cognition)} = k * \text{cpu-time(cognition)}$$

and, from the previous expression,

$$\text{elapsed-simulation-time(fire)} = k * \text{cpu-time(cognition)}$$

The advantage of this proportionality is that we now have a way to exert time pressure on cognition. The *real-time knob* is the device that exerts pressure, simply by increasing k. Clearly, we can change k without changing the amount of cpu time allocated to cognition, and when this happens, the net effect is to increase the amount of simulation time allocated to the fire. Because of the strict proportionality between simulation time and cpu time for cognition, the indirect effect of increasing k is to *reduce the amount of simulation time allocated to cognition, relative to the simulation time allocated to the fire.* That is, to increase time pressure on cognition. Currently k = 300, which means that one second of cpu time for cognition is matched by five minutes of simulation time for the fire. If we increase k to 600, then the fire is allowed to burn for 10 minutes for every cpu second of cognition time.

Cognitive tasks are allotted a full synchronization quantum each time they run. At times there are not enough cognitive activities to fill a quantum, in which case the task ends and waits to be rescheduled. Some cognitive activities take longer than a full quantum, in which case their internal state is saved between quantum steps.

Example: Imagine it is now 12:00:00 in the simulated world, and an agent is about to begin planning. After one cpu second, simulation time for the agent is 12:05:00. The fire is thus "owed" five minutes of simulation time. But before it runs, the DES runs all sensor, effector, and reflex tasks. After that, it may take 7 cpu seconds to calculate the effects of five minutes of fire. Moreover, simulation time is still 12:05:00, because the agent and the fire are simulated parallel processes. So after roughly eight cpu seconds (one for the planner, negligible time for sensors, effectors, and reflexes, and seven for the fire), we have simulated five minutes of planning and five minutes of fire, and both processes are paused at 12.05.00.

3.3 Instrumentation of the Testbed

The Phoenix testbed is designed to support experiments with a variety of IRTPS architectures---not only our Phoenix agent architecture. Currently it has been instrumented to some extent, and much more instrumentation is planned. In this subsection we describe three levels of instrumentation suggested by Nort Fowler. Low level metrics are largely hardware-dependent estimates of how the software system is utilizing the hardware. Middle level metrics give us a fine-grained picture of how a specific architecture behaves over time; for example, we can measure the communication overhead among agents, the time required to respond to significant changes in the environment, the amount of time spent in error recovery, the ability of scheduling algorithms to meet deadlines, and so on.are specific to the agent architecture. High level metrics are domain specific. They record features of the environment that are affected by the agents, such as acreage burned by fires, and consumption of resources.

Any researcher who implements a new agent architecture in the Phoenix environment will have to define middle-level metrics, because these are architecture specific, but probably won't have to define high and low level metrics. For example, Phoenix agents maintain a timeline of pending actions, and we need to know the average latency between posting and executing an action. An agent architecture implemented as a blackboard system may instead look at the scheduling of tasks on an agenda. Most of our middle level metrics are for the Phoenix agent architecture, not for unanticipated other architectures.

Currently, the following instrumentation is complete or nearly so:

Low Level Instrumentation

Run time , Cpu time , Disk wait time, Time since last run , Idle time, Utilization, Overall. Each of these is graphed against time.

We also provide an interface to the Explorer performance metering tools which work at the function call level and provide for each function the: Number of calls , Average run time, Total run time, Real time, Memory allocation, Page faults

Middle Level Instrumentation

Statistics on cpu utilization by the cognitive component of each agent to see the actual profile of real-time response, graphed against time

The latency between when actions on the timeline become available for execution and when they are executed.

Metrics on sensor and effector usage

Metrics on reflexes

The goal of these metrics is to compare the utilization of cognitive and other resources in different scenarios (see Section 3.4). Each scenario will contain important events (e.g., a fire is detected). We will graph these metrics against time and annotate the graphs at the points that the significant events occured, so we can see how the agent architecture responded.

High Level Instrumentation.

Fire destruction is currently measured by amount and type of forest, houses, and agents burned.

Resource allocation is currently measured by amount and type of agents employed to fight the fire, gasoline consumed, fireline cut, distance traveled, and time required to contain the fire.

3.4 Baseline Scenarios

One advantage of studying IRPTS in a simulated environment is the ability to run the same environmental scenario again and again while modifying aspects of the agent architecture (see Section 5). We have recently implemented the ability to define *scripts*, which include the type and number of available agents, and guide the environment through a series of changes in conditions such as windspeed and other weather conditions. We also have the ability to introduce stochastic factors into scripts, such as lightning strikes. Besides scripts, we will soon be able to provide baseline statistics on events such as rates of spread of fires in different conditions.

Scripts play an important role in evaluating IRTPS systems, and comparing IRTPS architectures. By design, scripts can force an agent to confront virtually any IRTPS issue. Here is a simple script that raises the six IRTPS issues that were discussed in the original IRTPS Initiative:

Materiel:

One fireboss to coordinate the activities of three bulldozers

Bulldozers can move 6.5 kph in softwood, 56 kph on road, .5 kph while building line

One watchtower at location approx 42500x37500

Environment:

A fire of radius 700 km starting at coordinates approx 45000x46250,

Starting wind speed and direction 3 kph from the south

Environmental Changes:

At time 2 hr, wind changes to 10 kph from the NW, threatening buildings---the base and lodge

Since it involves a burning fire, the script requires agents to produce relevant output in a timely fashion. The particular scenario includes an environmental

change (asynchronous with the reasoning system) that invalidates previous input, necessitating the detection of a new threat to higher priority areas and a redirection of ongoing reasoning in order to protect them. To handle this scenario, a system must reason efficiently and effectively about temporal processes, namely the expected progress of a fire under particular environmental conditions and the abilities of a limited number of agents to take steps, over time, toward putting out the fire.

We must add that this script confounds the current implementation of Phoenix agents. They are currently incapable of redirecting their efforts to save the base and lodge.

3.5 Portability.

The Phoenix system runs on color and monochrome Texas Instruments Explorers and MicroExplorers. We can package everything together (including support-code for the frame system, grapher, EKSL utilities, etc.) as needed. We are making progress on the documentation.

The entire Phoenix system is designed to be modular, so fellow researchers can use the components they want. The smallest self-contained module, and the most basic, is the Phoenix environment. This includes the DES, the Map layer, and the user interface. We are confident that a researcher could take this code and build his or her own agents to interact with it. However, we have not done this in our own lab, so we cannot be sure.

Above the environment are three additional levels of software---the Phoenix basic agent architecture, our own Phoenix agents, and the organizational structure that holds among our Phoenix agents. The basic agent architecture is a skeleton with hooks for sensors and effectors, reflexes, and a cognitive component (see Section 4). Some weeks ago the entire lab went through the exercise of defining a new type of Phoenix agent (an airplane) given only the basic agent architecture, and we are confident other researchers can do the same. There are really two aspects to defining a new agent. One is mostly bookkeeping: We define frames for the agent that describe its physical abilities, so that the DES knows how it behaves over time. We also define frames that add instances of the new agent to a script. This is the easy part. The hard part is defining the cognitive abilities of the agent. In terms of the basic Phoenix agent architecture, this means defining plans, execution methods, a cognitive scheduler, and other architectural components discussed in Section 5.

Of course, the Phoenix environment does not and should not care about the cognitive component of a new agent, other than to schedule its processes to guarantee the illusion of simultaneity with other agents and environmental

processes. Thus, it is relatively easy to tell the Phoenix environment about the physical abilities of new agents, as in the examples above, and unnecessary to tell the environment how the cognitive components of the agents work. We hope this will make it easy for researchers to use the Phoenix environment, or the environment and the basic agent architecture, to design and test their own agents.

4. Toward a Solution: The Phoenix Agent Architecture

A uniform agent architecture is shared by all agents. This architecture is the structure of the agent, the "hardware" that dictates the fundamental faculties and limitations of the agent. The structure endows and bounds acuity, speed of response, and breadth of action. The structure constrains what an agent can do, but not what it does. Specific methods control what the agent does. Control methods determine what to do and how to do it. This dichotomy between structure and control is reflected in this subsection and the one following it. Section 4.1 describes the agent architecture and Section 4.2 focuses on techniques for real-time problem solving. For a more detailed description of these components, see [Cohen, 1989])

4.1 Phoenix Agent Architecture

The agent architecture has four components. *Sensors* perceive the world. Each agent has a set of sensors, such as fire-location (are any cells within my radius-of-view on fire?) and road-edge (in what direction does the road continue?). *Effectors* perform physical acts such as moving or digging fireline. *Reflexes* are simple stimulus-response actions, triggered when the agent is required to act faster than the time-scale for the cognitive component. An example is the reflex of a bulldozer to stop if it is moving into the fire. The *cognitive* component performs mental tasks such as planning, monitoring actions, evaluating perceptions, and communicating with other agents. Although every agent has these components, each component can be endowed with a range of capabilities.

Sensors get input from the world (fire simulation and map structures). Their output goes to state memory in the cognitive component, and also to the reflexive component (triggering instant responses in the form of short programs to the effectors). For example, a bulldozer sensor that detects fire within its radius-of-view updates state memory automatically. If the detected fire is in the path of the bulldozer, the emergency-stop reflex is also triggered. Effectors are programmed by the cognitive component and by reflexes. Their output performs actions in the world. In the preceding example, the emergency-stop reflex would program the movement-effector of the bulldozer to stop. If the fire were not too close, the cognitive component might then step in and program the movement effector to start moving parallel to the fire. If the cognitive component also programmed the blade effector to put the blade in the down position, the bulldozer would not only

maintain a safe distance from the fire, but it would also build fireline as it moved. Sensors and effectors are first-class objects whose interactions with other components and the world are implemented in Lisp code. Reflexes, as mentioned, are triggered by sensory input, which causes them to program effectors to react to the triggering sensation. They are implemented in production-rule fashion, with triggering sensations as their antecedent clauses and effector programs as their consequents. Because they respond directly to the environment and so must keep up with it, sensors, effectors, and reflexes operate at the same time scale as the simulation environment and are synchronized as closely as possible within the discrete event simulator.

The cognitive component receives input from sensors and sends programs to the effectors to interact with the world. It is responsible for data integration, agent coordination, and resource management, in other words, most problem solving activity. This component operates in larger time slices than the others, thus reducing the overhead of context switching, but increasing the possibility of reasoning with outdated information.

The Phoenix cognitive component directs its own actions by adding prospective actions onto the timeline, a structure for reasoning about the computational demands on the agent, then selecting and executing these actions one at a time. Actions may be added in response to a change in environmental conditions (e.g., a new fire) or as part of the computation of other actions (e.g., through plan expansion). Every action that the cognitive component accomplishes is represented on the timeline with its temporal relations to other actions and resource requirements (e.g., processing time and necessary data). The cognitive scheduler decides which action to execute next from the timeline and how much time is available for its execution.

Actions may perform calculations, search for plans to address particular environmental conditions, expand plans into action sequences, assign variable values, process sensory information, initiate communication with other agents, or issue commands to sensors and effectors. These actions are represented in skeletal form in the plan library. Actions are described by what environmental conditions they are appropriate for, what they do, how they do it (the Lisp code for their execution, called the execution methods), and what resources and data, environmental and computational, they require. A plan is a special type of an action. It includes a network of actions related by their data references and temporal constraints.

Planning is accomplished by adding an action to the timeline to search for a plan to address some conditions. When the search action is executed, it selects an action or plan appropriate for the conditions and places it on the timeline. If this new action is a plan, then when it is executed it expands into a plan by putting its sub-actions onto the timeline with their temporal inter-relationships. If it is an action, it instantiates the requisite variables, selects an execution method (there

may be several with differing resource requirements and expected quality of solution), and executes that method. We call this style of planning *skeletal refinement with lazy expansion.* Plans are represented as shells that describe what types of actions should be executed to achieve the plan but do not include the exact action or its variable values until it is executed. Delaying expansion allows the expanded plan to address more closely the actual state of the environment during execution.

 This planning style is common to all agents in the Phoenix planner, though it is flexible enough so that agents with a variety of cognitive capabilities are possible. For example, the fireboss has far more sophisticated methods for gathering and integrating information than the bulldozer does. It can direct the actions of the bulldozers, while the bulldozers can only make requests of the fireboss. However, the fireboss, unlike the bulldozers, does know how to get out of the way of the fire because it does not work close to the fire.

Creating a different type of agent requires defining a cognitive component. One can optionally define a set of programmable sensors and effectors (of arbitrary complexity) and add a set of reflexes to handle situations that require instant response by the agent. To create sensors and effectors, the simulator must be told rates of action under varying environmental conditions, range of perceptions, and other physical capabilities. Creating reflexes involves describing the triggers, the expected output from sensors, and the response, the programming for the effectors. The default cognitive component consists of plans, which are networks of actions available to the agent and tailored to situations in the environment, and methods which describe how to execute the actions. Creating a new cognitive component with the same structure as that described here involves defining a new plan library.

Several design decisions in the Phoenix agent architecture have been made specifically to facilitate real-time control. One important decision is to incorporate both reflexive and cognitive abilities in agents, enabling agents to respond reflexively to events that occur quickly, while responding more deliberately to resource management and coordination problems on a longer time scale. The combination of a reflexive and cognitive component accounts for time scale mismatches inherent in an environment that requires micro actions and contemplative processing. Micro actions, such as following a road and keeping out of the immediate range of the fire, involve quick reflexes and little integration of data. Contemplative processing, such as route planning, involves long search times and integration of disparate data such as available roads, terrain conditions, and fire reports. This horizontal decomposition ensures that the agent can perform reflex actions to keep it from danger and maintain the status quo, while also performing more contemplative actions. This strategy for responding to disparate demands of the environment is advocated by Brooks, and Kaelbling; although in both cases, they chose more levels of decomposition for their domains. Our agent architecture, in effect, combines two different planning components:

one highly reactive, triggered by specific environmental stimuli and operating at very small time scale, and the other slower and more contemplative, integrating large amounts of data and concerned with resource management and coordination.

Another design feature that facilitates real-time control is the timeline and its single representation for all actions. Because prospective actions share a uniform representation on the timeline, all problem solving actions have access to the same memory structures and can be monitored and allocated resources using the same mechanisms. All problem solving tasks are subject to the same constraints with respect to resource allocation: how much time is required, what information gathering resources are required, and what data is necessary. This framework allows new cognitive capabilities to be integrated easily by defining their requirements within the action description language and relying on the timeline and its supportive scheduling mechanisms to temporally arbitrate their allocation.

Lazy skeletal expansion also facilitates real-time control. Plans are only partially elaborated before the agent acts. This deferred commitment exploits recent information about the state of the world to guide action selection and instantiation. Completely deferred commitment, such as in reactive planning, is probably not tenable when agents or actions must be coordinated or scarce resources managed. The integration of planning and acting in Phoenix is designed to be responsive to a complex dynamic world by postponing decisions on exactly what action to take, while also grounding potential actions in a framework (skeletal plans coordinated on the timeline) that accounts for data, temporal and resource interactions.

4.2 Real-Time Control in the Agent Architecture

How does a Phoenix agent respond to real-time pressure? One approach is to control processing requirements. This enhances the flexibility of actions and the sophistication of control decisions. Providing alternative execution methods for timeline entries ensures a range of choices that vary in their timeliness. Different scheduling strategies for managing the actions on the timeline provide greater responsiveness to real-time constraints. Another approach is an expectation-based monitoring technique that reduces the overhead of monitoring while providing early warning of plan failure. Earlier warning of plan failure affords the planner more time to adjust and more flexibility in possible responses. These approaches are discussed below.

4.2.1 Control of Processing Requirements

Processing requirements can be controlled in two ways: by controlling how much time is used by individual actions and by controlling the overall distribution of time across all actions. Approximate processing and anytime algorithms are methods

for controlling how much time is used by individual actions. In these methods, processing time is traded against quality or correctness of solution to satisfy time constraints that could not be managed under rigid processing demands. In Phoenix, these methods are alternative execution methods. Execution methods, as introduced in Section 4.1 are lisp code that performs the cognitive actions. Each cognitive action may may be executed by one of several execution methods, with differing time requirements and so differing solution expectations. The Phoenix planner delays the choice of an action's execution method until the cognitive scheduler selects the action for execution, thereby allowing the scheduler to select a method suited to existing time constraints. By postponing the ultimate commitment of cognitive resources until a choice must be made, those resources can be allocated judiciously.

Alternative execution methods are particularly useful in actions that incur potentially high computation costs with predictable results, such as path planning. Phoenix uses an A* algorithm to calculate paths for bulldozers. It searches the two-dimensional map representation of the world for the shortest travel time path between two points. It expands the current best path incrementally, searching each unobstructed neighboring cell for the best next step. The algorithm is parameterized to work at multiple levels of resolution, so that search steps could range from 128 meters up to 8 kilometers. A small search step, 128 meters, yields the shortest path, requiring the least travel time for the bulldozer. However, this resolution requires the most computation (i.e., cognitive resources). The largest search step, 8 kilometers, typically yields a longer path, which requires more travel time, but can be calculated quickly, consuming less computation time. At times it even fails to find a solution, since there are bottlenecks in the map that don't appear at large search steps. Each of these resolutions constitutes a different execution method for calculating a path, alternative methods which trade-off cognitive-time for quality of solution.

The cognitive scheduler controls the overall distribution of cognitive processing time across all actions. At each time step, it selects the next action from the timeline to execute, chooses an execution method for the action, and executes it. Thus, the scheduler is key to controlling the responsiveness of the cognitive component to real-time constraints. The current version of the scheduler for Phoenix is rudimentary and considers only a short horizon for scheduling decisions. It selects the next action for execution based on timeline ordering, action priority and the amount of time an action has been waiting for execution. A more sophisticated scheduler is being designed now.

4.2.2 Sophisticated Monitoring Through Envelopes

Just as we can explicitly represent the movements of an agent through its physical environment, so can we represent its movement through spaces bounded by failure or other important events. These spaces are called *envelopes*. Typically, one dimension of an envelope is time, and the others are measures of progress.

For example, imagine you have one hour to reach a point five miles away, and your maximum speed is 5 mph. If your speed drops below its maximum, for even a moment, you fail. As long as you maintain your maximum speed, you are *within your envelope*. The instant your speed drops below 5 mph, you *lose* or *violate* your envelope. This envelope is *narrow*, because it will not accomodate a range of behavior: any deviation from 5 mph is intolerable. Most problems have wider envelopes. Indeed, real time systems should be designed to ensure that narrow envelopes are the exception, not the rule.

The following problem illustrates a wider envelope. A bulldozer has one hour to travel five miles, as before, but its maximum speed is 10 mph. It starts slowly (perhaps the terrain is worse than expected). After 40 minutes it has travelled just two miles. It can still achieve its goal, but only by travelling at nearly maximum speed.

Clearly, if the agent waits 40 minutes to assess its progress, it has waited too long, because an heroic effort will be required to achieve its goal. In Phoenix, agents check their envelopes at regular intervals, hoping to catch problems before they get out of hand. One near-term research goal is to develop a theory of envelopes that will tell us when and how often they should be checked.

Agents check *failure* envelopes, which tell them whether they will absolutely fail to achieve their goals, and *warning* envelopes, which tell them that they are in jeopardy of failure. Typically, there is just one failure envelope but many possible warning envelopes. To continue the previous example, the bulldozer would violate a warning envelope if its average speed drops below 5 mph, because this is the speed it must maintain to achieve its goal. Violating this envelope says, "You can still achieve your goal, but only by doing better than you have up to this point." These concepts are illustrated in Figure 3 . The failure envelope is a line from "30 minutes" to "five miles," since the bulldozer can achieve its goal as long as it has at least 30 minutes to travel five miles. The average speed warning envelope is a line from the origin to the goal, but the bulldozer violated that envelope immediately by travelling at an average speed of 3 mph. In fact, it moved perilously close to its failure envelope. The box in the upper right of Figure 3 illustrates that the agent can construct another envelope from any point in its progress. In this example, the new envelope is extremely narrow.
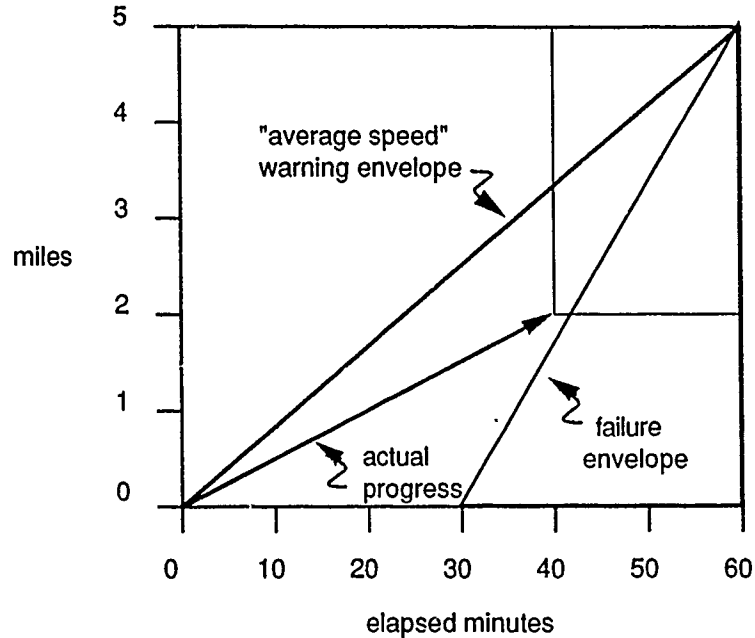
Figure 3. Depicting actual and projected progress with respect to envelopes

Agent Envelopes and Plan Envelopes. We distinguish between the envelopes of individual agents and those of multi-agent plans. In Phoenix, plan envelopes are maintained by the fireboss agent, who coordinates several subordinate bulldozers. Because the environment changes, global plans may be put in jeopardy even if agents are making progress that, from their local perspective, is well within their envelopes. Figure 4 illustrates plan envelopes as they are currently implemented in Phoenix: The leftmost illustration represents the current state of the fire, its projected boundaries after one and two hours, and the firelines that three bulldozers are expected to cut. By projecting where the fire will be, then adding some slack time, the fireboss anticipates that the last of these lines will be cut an hour before the fire reaches it. On the right of Figure 4, we see the actual progress of the fire: After one hour, it has grown less than expected, so the amount of slack time grows (bottom of Figure 4) and the plan stays well within its one hour slack time envelope. But during the next hour, the fire grows more rapidly than expected; so rapidly, in fact, that the slack time envelope is violated. Sometime during this interval, the fireboss will check the plan envelope and discover that it is violated. It then replans and typically sends one or more additional bulldozers to help out.
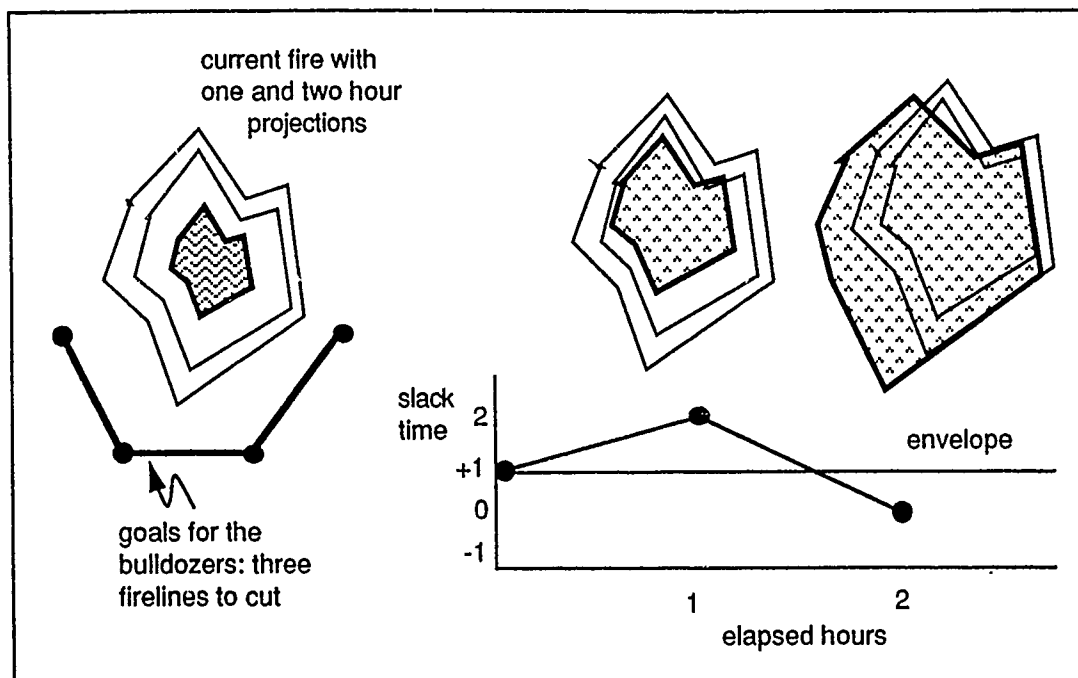
Figure 4. A plan envelope for maintaining slack time.

The Utility of Envelopes. A planner can represent the progress of its plan by transitions within the plan's envelopes. Progress, failures and potential failures are clearly seen from one's position with respect to envelopes, whereas this information is not always apparent from one's position in the environment.

Envelopes function as early warning devices in two ways. First, explicit warning envelopes alert the planner to developing problems. Second, failure envelopes can tell an agent it has failed long before its allocated time has elapsed. In Figure 3, for example, the agent knows it has failed as soon as it crosses the envelope. A third kind of early warning has yet to be implemented: Just as a planner can project the course of events in its environment, so it can project its progress within its envelope and, particularly, when an envelope might be violated. A simple projection method is extrapolation. For example, if we checked the envelope in Figure 4 after 75 minutes we would see a "downward" trend. By linear extrapolation we could estimate when the envelope would be violated. Of course, the downward trend may reverse, or level out. But sometimes it will be worthwhile to have the projected time of envelope violation despite its uncertainty.

Envelopes integrate agents at different levels of a command hierarchy: A fireboss agent formulates a goal and a corresponding envelope, and gives them to a subordinate bulldozer agent with the following instructions: "Here is the goal I want you to achieve. I don't care how you do it, and I don't want to hear from you unless you achieve the goal or violate the envelope." The bulldozer then works independently, not monitored by the fireboss. It figures out where to go, how to avoid obstacles, and how to keep clear of the fire, until its goal is achieved or its envelope violated. Meanwhile, the fireboss is free to think about other agents, other

goals, or to replan if necessary. Envelopes grant subordinate agents a kind of autonomy, and grant superordinate agents the opportunity to ignore their subordinates until envelopes are violated.

We have yet to develop cognitive scheduling mechanisms to take full advantage of envelopes. The design of these mechanisms is motivated by the following questions: How often should envelopes be checked? Should we adopt a fixed interval or a dynamic one, and if the latter, what execution methods will determine when to check next? When should agents project envelope violations and how should they use the projections? Given that checking a plan envelope, or projecting progress with respect to it, may involve collecting and integrating information from the environment and all the participating agents, the cognitive overhead of of these activities can be considerable and must be carefully scheduled.

5. Methodological Issues.

Our overriding research goal is to develop a sound basis for the design of AI agents. AI is a kind of design. We don't design graphics, or VLSI circuits, or mechanical devices: we design intelligent agents. The agents are evaluated by how they behave. Their behavior is determined by their environments and their architectures. Once we adopt this view, we see immediately that we do not know enough about the relationships between agent architectures, behaviors, and environments (the corners of the behavioral ecology triangle in Fig. 1) to design intelligent agents in a principled way. For example, we cannot even precisely define the characteristics of environments (Sec. 2.2), much less behaviors. And we cannot answer the question, "How would the behavior of this AI program, in this environment, change if you change its architecture this way: ... ?" But until we can answer this question, AI system design will remain ad hoc.

In fact, design is one of six research activities implied by the behavioral ecology model. Here is the complete list:

Prediction: How will behavior be affected by changing the architecture of the agent or its environment? For example, how will behavior be affected by changing the size of short-term memory, or by changing the mechanism by which long term memory is accessed? How will behavior be affected if the environment "speeds up," so that events that took N seconds now take N/2 seconds?

Explanation: Why does a particular behavior (presumably unexpected) emerge from the interaction between an agent and its environment? For example, why does an agent that combines long-term, goal-directed behavior with short-term reactive behavior sometimes exhibit something like an approach-avoidance conflict---dashing first toward a goal, then away from it, but getting nowhere in the long run?

Design. What architectures will produce a particular set of behaviors in particular environments? For example, what architectures will enable an agent to respond to events in the environment that occur at very different time scales?

Environment analysis: What aspects of the environment most constrain agent design? What is our model of the environment?

Generalization: Whenever we predict the behavior of one agent in one environment, we should ideally be predicting similar behaviors for agents with related architectures in related environments. In other words, our theories should generalize over architectures, environmental conditions, tasks, and behaviors.

Functional relationships: What knowledge do we need to answer questions in these classes? What are the functional relationships between the architecture of an agent and its behavior?

Both the behavioral ecology model and the S/E model of Rosenschein, Hayes-Roth, and Erman (see their paper in this volume) explicitly acknowledge the relationships between architecture the environment, and behavior. Rosenschein et al. denote the architecture and environment S and E, respectively; and characterize behavior as a sequence of state changes called a run. Furthermore, Rosenschein et al. seem to implicitly subsume, in what they call measurement and evaluation, some of the research activities above. But because neither the S/E model nor the behavioral ecology model make predictions, it is premature to compare them except to note some apparent differences in emphasis.

Rosenschein et al. view the "S/E boundary" as flexible, so that sometimes the environment can be made responsible for an activity that, in other circumstances, we might require of the agent. For example, with the general vision problem currently unsolved, we might construct an environment that "preprocesses" sensory data for the agent, thus moving the S/E boundary inward, toward the agent, bypassing the need for sophisticated sensors. This example suggests a small apparent difference between the S/E model and the behavioral ecology model: whereas the S/E model seems to assume a simulated environment, the behavioral ecology model does not. Although the Phoenix project uses a simulated environment, our principal research tasks (prediction, design, explanation, etc.) do not *presume* a simulated environment. It isn't clear yet whether the principal research tasks of Rosenschein et al. presume a simulated environment.

This raises the methodological question of whether one should use simulations at all. Some researchers insist that the subtleties of real environments are "lost in translation" to simulated environments. This is to some extent a straw man, because we don't view simulations as accurate representations of the real world. (In fact, we recently got into trouble by claiming that the Phoenix environment is an accurate simulation of forest fires.[4]) But it is important to distinguish *realism*

---

[4]See Letters to the Editor, AI Magazine, Vol. 10 No. 4.

and *accuracy*. Realism is necessary for our research; accuracy is not. Here are some examples of the distinction: In a realistic simulation, processes become uncontrollable after a period of time; in an accurate simulation, the period of time is the same as it is in the real world. In a realistic simulation, agents have limited fields of view; in an accurate simulation, agents' fields of view are the same as they are in the real world. In a realistic simulation, the probabilities of environmental events such as wind shifts are summarized by statistical distributions; in an accurate simulation, the distributions are compiled from real-world data. When possible, we use accurate data; for example, in Phoenix we use Defense Mapping Agency data of elevation, ground cover, and so on, and the fire dynamics are derived from U.S. Forest Service manuals (NWCG Fireline Handbook, 1985). But the goal of our research is not to accurately simulate forest fires in Yellowstone National Park. It is to understand the design requirements of agents in realistic environments—environments in which processes get out of hand, resources are limited, time passes, and information is sometimes noisy and limited.

With this in mind, we see that simulations have several advantages:

Control. Simulators are highly parameterized, so we can experiment with many environments. For example, we can change the rate at which wind direction shifts, or speed up the rate at which fire burns, to test the robustness of real-time planning mechanisms. Most important, from the standpoint of our work on real-time planning, is the fact that we can manipulate the amount of time an agent is allowed to think, relative to the rate at which the environment changes, thus exerting (or decreasing) the time pressure on the agent.

Repeatability. We can guarantee identical initial conditions from one "run" to the next; we can "play back" some histories of environmental conditions exactly, while selectively changing others.

Replication. Simulators are portable, and so enable replications and extensions of experiments at different laboratories. They enable direct comparisons of results, which would otherwise depend on uncertain parallels between the environments in which the results were collected.

Variety. Simulators allow us to create environments that don't occur naturally, or that aren't accessible or observable.

Interfaces. We can construct interfaces to the simulator that allow us to defer questions we'd have to address if our agents interacted with the physical world., such as the vision problem. We can also construct interfaces to show things that aren't easily observed in the physical world; for example, we can show the different views that agents have of the fire, their radius of view, their destinations, the paths they are trying to follow, and so on. The Phoenix environment graphics make it easy to see what agents are doing and why.

Let us return now to the comparison of the S/E and behavioral ecology models. We noted that the former model represents behavior as "runs," sequences of state transitions (or as measures over runs), whereas the behavioral ecology model is

inspecific about how to represent behavior. On the other hand, the behavioral ecology model is quite specific about the causal relationships that hold among the environment, the agent architecture, and the agent's behavior. The behavioral ecology model comes from biology; it regards the architecture as analogous to the genotype and the behavior as analogous to the phenotype. And it assumes that selection operates on the phenotype. Thus, there is no direct causal link between an agent's environment and its architecture; rather, the environment ensures that behaviors are differentially rewarded, so the architecture must be modified to produce "good" behaviors. This, then, is what we mean by a good architecture— one that produces behaviors that are good in a particular environment.

It's important to know whether such behaviors can be generated by design, that is, by intentional modifications to the architecture, or whether they must evolve by search. Advocates of emergent behavior often take the latter view. They say that one cannot generate the phenotype from the genotype; one cannot predict how a moderately complex architecture will behave. This has important practical and methodological implications for IRTPS. Do we build IRTPS systems "top down," by assembling components that are predicted to behave in particular ways, and damn the emergent behaviors? Or do we build them "bottom up," by assembling components incrementally and empirically, waiting for desired (and undesirable) behaviors to emerge? In fact, we mix the approaches in proportions determined by the degree to which behaviors can be predicted from architectures (or components of architectures). Moreover, this degree of predictability is determined in part by the desired precision or scale of the predictions. If you want to know the precise number of cpu seconds that a process will run, you are probably out of luck. But if you want to know the upper bound runtime, it may be possible. You probably can't know the exact location of a fire ten minutes from now, but you can certainly draw a circle that has a high probability of circumscribing the fire. Thus, the question of whether behaviors can be generated by design depends intimately on how precisely we want to specify and predict the behaviors.

This brings us to a final methodological issue: evaluation. Let us first ask, What is being evaluated? Whether an architecture exhibits "timely" behavior? or exhibits a good tradeoff among several desired behaviors? Whether the behaviors are exhibited in a sufficiently wide range of environments? Whether we can predict when the behaviors will and will not occur? Whether we understand the functional relationships between architecture and behavior well enough to design an agent that will exhibit desired behaviors in a new environment? All of these should be evaluated. More pointedly, evaluation *cannot* stop with the demonstration that a system "works," however sophisticated the demonstration! We must take at least two more steps. We must attempt to show *why* the solution works (or doesn't work). This is uncommon, but essential if we are to make progress as an engineering field. The third step is to show why any solution with such-and-such abstract characteristics must work (or not work). This requires models of the behaviors and environments under study.

5.1 An Example of Design for IRTPS

We will briefly illustrate the previous points, and the terminology in Section 2.2, with the example of the design of Phoenix's cognitive scheduler. It's current cognitive scheduler is very weak. We are designing another one that achieves many of the behavioral goals of IRTPS. In the terms of Section 2.2, this seems to imply that we should list the problems and inherent problems, describe the relevent ECs and ACs, and after analyzing how the problems arise out of the interactions between ECs and ACs, we propose solutions and solution realizations. In fact, this seems to be an idealization. Instead we start with an *informal* description of some problems, and then hunt around for ECs and ACs that we believe account for the problems. The result is a formal description of the problems in terms of ECs and ACs. Then we generate solutions and solution realizations.

Here is an example of the first steps.

Informal description: The plan selection mechanism may take too long to find a plan. As a result, the fire may burn too much area, or may become uncontrollable. (Note that this is intentionally vague, to show how we formalize the problem description in terms of ECs and ACs.)

Environment characteristics: What is going on in the environment that could contribute to the problem, as informally described above? Let's concentrate on one thing, the spread of the fire. We want to model this in a way that allows us to firm up the informal problem description. Suppose we model the spread of the fire as an exponential process analogous to compound interest and population growth. Then, the perimeter of a fire after t time units is:

$$p = p_i(1 + r)^t$$

Here, p is the perimeter of the fire, p_i is the initial perimeter of the fire, r is the percentage increase in the fire perimeter every time unit, and t is the number of time units that have elapsed.

Architecture characteristics. For now, we will list just two characteristics: It takes a period of time, d, to generate a plan and get the bulldozers to the fire to begin implementing the plan. And once at the fireline, bulldozers dig at a constant rate.These are both oversimplifications, but useful, as we shall see.

Now we can say more formally what the problem is:

length of perimeter

length of
controlled
perimeter

p'

case A

case B

d
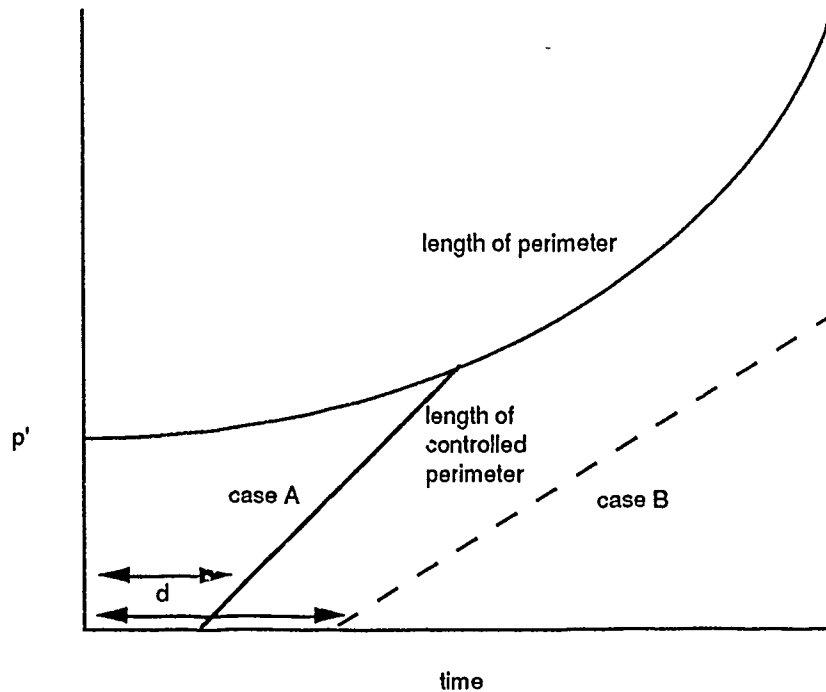
time

Figure 5

The curved line represents p, the length of the perimeter, as a function of time. It begins not at the origin, but at a point that represents the initial perimeter of the fire (e.g., its size when detected). We assume for simplicity that the steepness of the curve is described by one parameter, r, which captures factors such as wind speed and fuel type. Obviously, a more complex model could be generated if needed. After some delay, d, a plan is detected and some bulldozers are dispatched and then arrive at the fire. They begin cutting fireline at a constant rate, so the length of the controlled ·)erimeter increases at a linear rate determined by the number of bulldozers. We show two possibilities, case A and case B.. In case A (solid line) the bulldozers arrive at the fire sooner, and in greater numbers than in case B (dashed line). In fact, in case A the fire is controlled, whereas in case B it is not. We know this because the line for case A intersects the line for the perimeter, which means that at some point, the length of the controlled perimeter equals the length of the fire perimeter; or, all the fire perimeter is controlled. In case B, this doesn't happen.

Before we can rephrase the informal problem description more precisely, we need to know what affects the parameters represented in the diagram above. This will tell us what we control as designers, what the system itself controls as an autonomous agent, and what the environment alone controls.

p' — This could be reduced if the fire was sighted earlier. The lower limit on the speed with which the fire is sighted is an AC that we control. It depends on things like how big a fire must be before it is noticed, how often the watchtowers look, how long it takes them to report their findings, how long it takes the fireboss to notice, etc. Most of these ACs have lower limits that we control, and actual values that the agent controls.

r — This parameter, which determines the steepness with which the perimeter increases, is an EC.

d — as with p',we control the lower limit on d, and the actual value is controlled by the agent.

slopes of "controlled perimeter" lines — this has an upper limit that we control (by controlling the number of available bulldozers) and an actual value that the agent controls, by controlling the number of bulldozers committed to the fire. .

At this point, we can begin to give formal descriptions to problems. For example, what is a *deadline*? In general, a deadline is a point at which the value of problem solving changes, usually downward. Consider a hard deadline for the plan selection process. In the previous diagram, this is represented as an upper limit on d. Consider three cases, denoted c, d, and e in the following diagram:
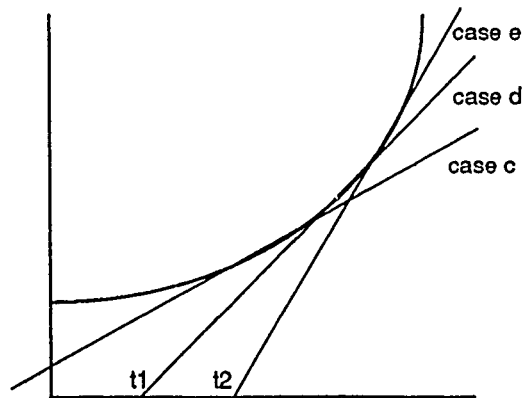
In case c, the slope of the "controlled perimeter" line is shallow because, say, it corresponds to a plan that involves only two bulldozers. Moreover, the deadline for the institution of the plan *has already passed*. You can see this by shifting the line for case c to be tangent to the "perimeter" line. By the same operation you can see that, for case d to succeed, the bulldozers (more of them than in case c, hence the steeper slope) must be at work before t1; and for case e to work, they must be busy by t2. Any delay longer than t1 or t2 shifts the respective lines to the right, ensuring that they will never intersect the perimeter line and the fire will never be controlled.

Now we can be more precise about the problem: For given values of r and p' (assuming the fire has just been sighted), find a plan that is expected to contain the fire and that can be instituted before its deadline.

Note that the original problem, a failure to get plans ready in time, has been formalized in the context of an agent model and an environment model. Moreover, a common IRTPS term has been defined in these contexts. One might argue that, in the process, we have taken a nice, general term like "deadline" and replaced it with something that is so specific to Phoenix as to be unusable. We believe we have done exactly the opposite. Not only have we made a vague term precise, but we have also identified a very general functional relationship or "rule" associated with the term: Imagine that the perimeter of the fire grows linearly, not exponentially. Then the notion of deadline illustrated in Figure 5 would not exist. If a process F grows linearly, and another linearly-growing process B is trying to control it, then a comparison of the growth rates of F and B will tell us whether B will succeed, and when it will succeed (assuming the growth rates don't change). If B grows faster than F, then it will control F eventually. The only effect of delaying the onset of B is to delay the control of F. On the other hand, if F grows superlinearly, as in Figure 5, and B grows linearly, then a delay does not merely delay the event in which B controls F, it may make that event impossible (as shown by the dashed line in Figure 5). we believe this is a very general phenomenon, and thus a very general interpretation of "deadline": A deadline is the point at which a linear process becomes incapable of catching—at any time in the future—a superlinear one. Obviously this can be generalized to functions of other orders—a sublinear process trying to catch a linear one, and so on.

As we evaluate the Phoenix project, we will certainly ask whether it plans in a timely way, whether it meets deadlines, balances cognitive load, exhibits graceful degradation, and so on. But the most telling evaluation will be whether we have been able to engage in the specialization-generalization process illustrated above: Whether we gave terms such as "deadline" precise interpretations in terms of Phoenix ECs and ACs and then generalized them again, as we did when we said a deadline is a point at which one process becomes incapable of catching another.

References:

Cohen, P. R., Greenberg, M. L., Hart, D.M., Howe, A. E. 1989.Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. AI Magazine, Vol. 10, No. 3. pp. 32-48. Fall, 1989.

Cohen, P.R. and Howe, A. E. 1988. How Evaluation Guides AI Research. AI Magazine 9(4) 35—43.

NWCG Fireline Handbook 410-1, National Wildfire Coordinating Group, Boise, Idaho, Nov 1985.

Rosenschein, S. J., Hayes-Roth, B., and Erman, L. D. Notes on Methodologies for Evaluating IRTPS Systems. In this volume.

The Phoenix Testbed

[Draft Version]

Michael Greenberg   David L. Westbrook

February 21, 1990

Contents

Chapter 1

Introduction

The Phoenix testbed[1] contains four components: A task model, a map representation, a user interface and a fire simulation. In this document, each component is described conceptually, what is it and how does it work, and functionally, how is the component used.

The appendices contain a description of the file organization, and installation and maintenance instructions.

In a view, the testbed is an environment for simulating processes that need to be synchronized in *time*. Each process is called a *task*. In addition to tasks, the testbed provides a topological representation of the world. This *map* contains information about vegetation, roads, rivers and buildings. A user interface and a forest fire simulation are also provided.

1.1 Simulation and Time

Phoenix provides for the simulation of the world by allowing the user to define tasks which "run the world." By maintaining a global (between all tasks) notion of time, the system is able to control each task to make sure that the tasks stay synchronized. Thus, each task can implement a part of the world, and Phoenix makes sure all the parts are kept up to date. Tasks communicate via shared data structures and system defined synchronization methods.

A simple example will make this clear. Suppose we want to simulate a world that has forest fires and firefighters. We can define a task which burns the fire (the simulator), and one task for each firefighter. The tasks must be synchronized by some global notion of time. In a five minute period, for instance, the fire can burn only so far and a firefighter can only do a certain amount of problem solving. The tasks interact by modifying a global data structure, in this case the map of the world. The simulator burns things, and the firefighters try to contain the fires.

The *Phoenix Scheduler* is responsible for keeping tasks synchronized. The global clock is called *simulation-time*. Each task must provide a method which specifies how simulation-time passes as it (the task) executes. There are three distinct types of time in the system:

Simulation Time This is the global shared clock to which each task is synchronized.

Task Time This is the time that each task thinks it is. Imagine that each task has a watch.

CPU time This refers to cpu time for a single task.

---

[1] The testbed is implemented in Common Lisp on the Explorer II, a high-performance standalone Lisp workstation.

Unless otherwise specified, all time units are measured in *internal time units*. Functions are provided to convert between time units (for example, minutes->internal-time and internal-time->seconds).

1.2   The Fire-system

While Phoenix is running, one object handles all requests by tasks, namely, the current instance of the fire-system flavor. The fire-system flavor contains the user interface, task scheduler, sets of tasks and the real world representation. It also provides numerous methods for manipulating those objects. The function fire-system returns the current instance of the fire-system flavor, which we will call that the current fire system.

Chapter 2

Tasks

Tasks are the organizing component of the Phoenix testbed. A *task* is a process plus a time-keeping mechanism. Each task must specify how *simulation-time* passes as the task's process executes. For example, a task may say "One second of cpu time corresponds to five minutes of simulation time." Three types of tasks exist; they differ by the time-keeping mechanisms they use.

CPU-time tasks. In a cpu-time task, simulation-time is a function of cpu time. When you define such a task, you specify the ratio of simulation time to cpu time. As the task runs, the Phoenix Scheduler keeps track of time appropriately. For example, the firefighting agents "think" via lisp code, and their "thinking speed" is set by some ratio of simulation time to cpu time, so that N cpu seconds of lisp execution is taken to be $C \cdot N$ seconds of simulated real time.

Explicit-time tasks. An explicit-time task is responsible for explicitly telling the scheduler how much time passed while it ran. For example, if there is a task to move an agent in the world, that task computes time as a function of speed and distance.

Periodic tasks. A periodic task runs at a fixed time interval. This interval is the task's *period.*

2.1 Tasks and the Scheduler

The tasks are interleaved on the cpu to simulate parallelism. The Phoenix Scheduler is responsible for allocating each task the appropriate amount of cpu time. The basic control structure of the scheduler is:

1. Select the task that should be executed next.

2. Start the task on the cpu.

3. When the task relinquishes control of the cpu, update its *task-time*. In general, cpu-time tasks relinquish control whenever there is a cpu timer interrupt (on the Explorer, this means a task gets at most one cpu second before relinquishing control). Explicit-time and periodic tasks explicitly return control to the task scheduler.

4. Repeat.

The scheduler updates each task's task-time by an amount computed based on the task's type. A simple example should illustrate this.

Suppose there are four tasks.

$Task_1$ is a periodic task that runs once every three minutes.

$Task_2$ is an explicit-time task.

$Task_3$ is a cpu-time task that runs at 5 minutes/cpu-second.

$Task_4$ is a cpu-time task that runs at 10 minutes/cpu-second.

When the simulation begins, the task-time for each task is zero. Following is a chronology of what happens:

> $Task_1$ executes at task-time 0 minutes. (This means that from the task's point of view, 0 minutes has elapsed since the simulation began). After it is done, the scheduler updates the task-time for $Task_1$ to be 3 minutes.

> $Task_2$ executes at task-time 0 minutes. When it relinquishes control, it reports that it used 7 minutes of simulation-time. The scheduler updates its task-time to be 7 minutes. Note that it is now "out of sync" with $Task_1$ by 4 minutes and $Task_3$ and $Task_4$ by 7 minutes.

> $Task_3$ executes at task-time 0 minutes. After using 1 cpu second, it is interrupted and the scheduler updates its task-time to be 5 minutes (1 cpu-second * 5 minutes/cpu-second).

> $Task_4$ executes at task-time 0 minutes. After using .8 cpu seconds, it is interrupted and the scheduler updates its task-time to be 8 minutes (.8 cpu-seconds * 10 minutes/cpu-second).

The scheduler maintains a *queue* of tasks sorted by task-time. The task chosen to execute next is the task at the head of the queue. The queue currently looks like this (the number is the task-time in minutes):

> $Task_1=3$, $Task_3=5$, $Task_2=7$, $Task_4=8$

> $Task_1$ executes for three minutes. When complete, its time is incremented by three minutes (the task's period).

> $Task_3=5$, $Task_1=6$, $Task_2=7$, $Task_4=8$

> $Task_3$ executes and gets .1 cpu-seconds. Its task-time is updated to 5.5 minutes.

> $Task_3=5.5$, $Task_1=6$, $Task_2=7$, $Task_4=8$

Notice that each task has its own idea about what time it is (that is, their watches disagree). This occurs for two reasons. First, we are simulating parallelism on a serial machine. Second, some tasks represent discrete processes. Given this, the tasks are all out of synchronization by some amount. The maximum "out of sync" time is the maximum over all tasks of *period, explicit time interval* and *max cpu quantum · simulation time / cpu time*. On the Explorer, the max cpu quantum is 1 second.

2.2   Defining and Using Tasks

This section describes how to define each type of task. Examples are given throughout. All tasks start with the task flavor, that is, to define a task you must first create a new flavor inheriting from task. You must also write a method which will *implement* the task and decide what the task's type is.

2.2.1 Defining CPU-time Tasks

To make a cpu-time task flavor called `generic-cpu-time-task`, do the following:

```
(defflavor generic-cpu-time-task ()
        (task)
      (:default-init-plist
       :initial-method :generic-cpu-time-task-toplevel
       :schedule-type :cpu-time
       :cpu-usec/internal-time  (round 1e6 (minutes->internal-time 5))))
```

This creates a task flavor named `generic-cpu-time-task` of type `:CPU-TIME`. The ratio of cpu-time to simulation-time is expressed in cpu-micro-seconds per internal-time unit. In the example, the ratio is set to 5 simulation-time minutes per one cpu-second. The method that executes when the task is run by the system is specified by `:INITIAL-METHOD`. In this case, the method `:generic-cpu-time-task-toplevel` must be defined. A cpu-time task normally shouldn't return control from the initial method once it is called. If it does, the task is *deactivated* (See section 2.2.4).

Normally, a cpu-time task is interrupted at least once every cpu-second (on the Explorer) and control is returned to the scheduler. If for some reason you want to explicitly return control to the scheduler, call `swap-in-scheduler`. Processing will continue from that point when the task is resumed.

2.2.2 Defining Periodic Tasks

To make a periodic task flavor called `generic-periodic-task`, do the following:

```
(defflavor generic-periodic-task ()
        (task)
      (:default-init-plist
       :initial-method :generic-periodic-task-toplevel
       :schedule-type :periodic
       :period (minutes->internal-time 5)))
```

This is almost identical to creating a cpu-time task. The differences are `:SCHEDULE-TYPE` and `:PERIOD`. When the scheduler decides to execute a periodic task, it executes the `:INITIAL-METHOD`. When the method returns, the task-time is incremented by the period, and the task is put back into the queue.

2.2.3 Defining Explicit-time Tasks

To make an explicit-time task flavor called `generic-explicit-task`, do the following:

```
(defflavor generic-explicit-task ()
        (task)
      (:default-init-plist
       :initial-method :generic-explicit-task-toplevel
       :schedule-type :explicit))
```

This is almost identical to creating a cpu-time task. Again, when the scheduler runs an explicit-time task, its `:INITIAL-METHOD` is executed. When the initial-method returns, the task is rescheduled (that is, reinserted into the task queue). Sometime during the execution of the method, the instance variable `restart-time` must be set to the appropriate time to run this task again. If `restart-time` is not reset, an error is signaled. If for some reason the task must restart again at the same time,

the :initial-method must return :RESTART-OK[1] At any time during its execution, an explicit-time
task can return control to the scheduler. To do so, it should set restart-time appropriately, then
call the function swap-in-scheduler. Processing will continue from the point immediately after the
swap-in-scheduler the next time the task is run.

2.2.4   The Task Life Cycle

There are several phases in the life of a task: creation, activation, execution, deactivation and termi-
nation.

Creation

The first thing to do is make an instance of a task with make-instance. When you create a task you
should give it a *handle*. A task handle is a name (represented as a symbol) that can be used to index
the task. You can also specify an :after :init method to perform instantiation-time actions.

```
(make-instance 'generic-cpu-time-task :handle 'task-1 :name "Task 1")
```

Activation

When a task is created, it is not available for execution. To make it available, you must *activate* it.

```
(tsend 'task-1 :activate)
```

Once activated, a task can be run by the scheduler. You can specify an :after :activate method.
The :activate method is run in the calling process, not the process associated with the task.
If you want a task to be activated at instantiation time, use the :ACTIVATE initialization option
(make-instance task ... :activate T)

The function tsend is like send, except that it takes a task-handle instead of a flavor object.

Execution

All active tasks can be scheduled by the Phoenix Scheduler. Each task runs in its own process.
Execution starts at the task's :INITIAL-METHOD. In a cpu-time task, the method should not return.
In periodic and explicit-time tasks, when the method returns, the scheduler may schedule another
task. These two types of tasks can assume that they won't be swapped until they either explicitly
release control, or the method returns. A cpu-time task may be interrupted any time, at any place in
its code.

Deactivation

A task can be stopped by *deactivating* it. Once deactivated, it is no longer eligible for execution by
the scheduler. To deactivate a task, send it a :deactivate message. A task can deactivate itself:

```
(send self :deactivate)
```

---

[1]INTERNAL NOTE: This will be changed to :RESTART-AT-SAME-TIME at some point. It is more intuitive.

or be deactivated by some other process

```
(tsend task-handle :deactivate)
```

You may specify a :before :deactivate method.

Reactivation

After a task has been deactivated, it may be reactivated again. When a task is reactivated, execution *always* begins from the initial-method. All internal state information may be lost. Activation and deactivation do not correspond to pausing and resuming. Activation is "reset to initial state and begin execution" and deactivation "stop and clean up." Since a task may be activated and deactivated many times (the UCL command Reset deactivates then activates all tasks), the :after :activate method should make sure the task's state is properly initialized, taking into account that the task may have been run previously.

Termination

At some point, you will want to kill a task. To do so, send the task a :kill message. A task can kill itself or be killed by some other process. Before a task is killed, it is deactivated if it is active. You may write :after :kill methods.

2.2.5 Managing Time

The current task-time is available to all tasks, and is returned by the function exact-time. In a cpu-time task, time is continuously changing, whereas in the other types, time changes in discrete steps. In cpu-time tasks, the scheduler computes the task-time.

For periodic tasks, task-time is updated by the task's period each time the initial method is executed. A periodic task may change its period by setting the period instance variable.

Explicit-time tasks are responsible for updating their own time before the initial method returns. This is done by setting the instance variable restart-time to be the next time the task is run. For example, an explicit-time task could say "Next time my initial-method is started, start it 5 minutes from now" as follows:

```
(incf restart-time (minutes->internal-time 5))
```

Each time an explicit-time task's initial method is started, the task's task-time is set to its restart-time.

All three types of tasks can say, "Put me to sleep for some amount of time." For explicit time and periodic tasks:

```
(incf restart-time (minutes->internal-time 5))
(swap-in-scheduler)
;; processing continues here 5 minutes later
```

For cpu-time tasks:[2]

---

[2]INTERNAL NOTE: A function should be written to do the right thing for all types of tasks. In fact, a set of functions should be written to handle task timing in a totally consistent (for all task types) manner. To date it hasn't been a problem because Phoenix does only simple timing. There should be things like sleep-for-time, wake-up-at-time, etc.

```
(setf cpu-time-adjustment (* cpu-usec/internal-time
                             (minutes->internal-time 5)))
(swap-in-scheduler)
```

2.3   A Real Example

This section contains an annotated example using four tasks that print informational messages. The function task-format prints a trace message to the trace pane in the 'process-display' screen configuration. The message automatically includes the task-time and the task's handle. So if the task called "T-1" at 1:00 P.M. on 8/1 executes

```
(task-format "Hello there")
```

the output is

```
[8/1 13:00 T-1: Hello there]
```

The first task will be a periodic task that wakes up once every three minutes. This task will keep track of the number of times it has run in the instance variable count. Notice that the :after :activate method sets count to zero.

```
(defflavor periodic-task (count)
       (task)
  (:default-init-plist
   :initial-method :periodic-toplevel
   :schedule-type :periodic
   :period (minutes->internal-time 3)))

(defmethod (periodic-task :after :activate) ()
       ;; When this task is activated, reset count to 0.
       (setf count 0))

(defmethod (periodic-task :periodic-toplevel) ()
       (incf count)
       ;; task-format prints a trace message
       (task-format "Count is ~d" count))
```

The second task will be an explicit-time task. This task also keeps track of the number of times it has been called with the instance variable count. Explicit-time tasks must keep track of their own time. The first time this task is run it will take one minute, the second time two minutes, then three minutes, and so on.

```
(defflavor explicit-task (count)
        (task)
  (:default-init-plist
   :initial-method :explicit-toplevel
   :schedule-type :explicit))

(defmethod (explicit-task :after :activate) ()
      ;; When this task is activated, reset count to 0.
      (setf count 0))

(defmethod (explicit-task :explicit-toplevel) ()
      (incf count)
      (task-format "Count is ~d" count)
      (incf restart-time (* count (minutes->internal-time 1))))
```

The third and fourth tasks will be cpu-time tasks. The difference between them will be their ratio of task time to cpu time. Since a cpu-time task's initial-method shouldn't return, a local variable can be used to keep track of iterations. The function 1-second-compute takes exactly one cpu second to execute (on an Explorer-II).

```
(defflavor cpu-task ()
        (task)
  (:default-init-plist
   :initial-method :cpu-toplevel
   :schedule-type :cpu-time))

(defmethod (cpu-task :cpu-toplevel) ()
  (do ((count 1 (1+ count)))
      (nil)
    (task-format "Count is ~d" count)
    (1-second-compute)))
```

To save time we have already defined these task flavors and methods in the file "*PH:TASKS;DOCUMENTED-TASK-EXAMPLE.LISP*". To run these tasks, load that file and start Phoenix.

Once Phoenix is up, select the 'process-display' screen configuration (user typein is shown after a Phoenix prompt).

```
Phoenix> process-display
```

Now we must create the four flavor instances:

```
(make-instance 'explicit-task :handle 'explicit-task :activate T)
(make-instance 'periodic-task :handle 'periodic-task :activate T)
(make-instance 'cpu-task :handle 'cpu-task-1
    :cpu-usec/internal-time  (round 1e6 (minutes->internal-time 5))
    :activate T)
(make-instance 'cpu-task :handle 'cpu-task-2
    :cpu-usec/internal-time  (round 1e6 (minutes->internal-time 10))
    :activate T)
```

The first cpu-time task runs at 5 minutes/cpu-second. The second runs at 10 minutes/cpu-second. The function create-task-demo-tasks (also defined in the *DOCUMENTED-TASK-EXAMPLE* file) creates all the instances, so:

```
Phoenix> (create-task-demo-tasks)
```

The simulation begins on August first at 12 noon. To run the system for 16 minutes, type

```
Phoenix> Run 16
```

When you start the system running, the scheduler will run each task at the appropriate time. You should see the output from the calls to task-format in the trace window. The output should start as follows:

```
[8/1 12:00 EXPLICIT-TASK: Count is 1]
[8/1 12:00 PERIODIC-TASK: Count is 1]
[8/1 12:00 CPU-TASK-1: Count is 1]
[8/1 12:00 CPU-TASK-2: Count is 1]
```

If everything works correctly, the explicit task will generate output at 12:00, 12:01, 12:03, 12:06 etc. Each time interval is one minute greater than the previous. The periodic task will count every three minutes. The first cpu-time task should count in five minute intervals, and the second at ten minute intervals. The output should continue with:

```
[8/1 12:01 EXPLICIT-TASK: Count is 2]
[8/1 12:03 EXPLICIT-TASK: Count is 3]
[8/1 12:03 PERIODIC-TASK: Count is 2]
[8/1 12:05 CPU-TASK-1: Count is 2]
[8/1 12:06 EXPLICIT-TASK: Count is 4]
[8/1 12:06 PERIODIC-TASK: Count is 3]
[8/1 12:09 PERIODIC-TASK: Count is 4]
[8/1 12:10 EXPLICIT-TASK: Count is 5]
[8/1 12:10 CPU-TASK-1: Count is 3]
[8/1 12:10 CPU-TASK-2: Count is 2]
[8/1 12:12 PERIODIC-TASK: Count is 5]
[8/1 12:15 EXPLICIT-TASK: Count is 6]
[8/1 12:15 PERIODIC-TASK: Count is 6]
[8/1 12:15 CPU-TASK-1: Count is 4]
```

When this runs, the output may vary slightly because when two tasks should run at the same time, the scheduler picks one arbitrarily.

Suppose you want to change the periodic task to run at 6 minute intervals. To do this you need to change the period and reset the system.

```
Phoenix> (tsend 'periodic-task :set-period (minutes->internal-time 6))
Phoenix> reset
```

If you run it again, the period will change. The Reset command sets the clock back to 12:00 and reactivates all the tasks. Note that if the :after :activate methods weren't specified, the count would continue from where it left off.

2.4   Input/Output

Since all tasks run as background processes, doing i/o is not straightforward. For all types of tasks, the easiest way to do output is with the functions task-format and debug-format. These functions are similar to format. The difference is that they generate an output message that automatically includes the task handle and task-time. The output is sent to the process-trace pane. The debug-format function also sends the output to the Phoenix Lisp Listener pane.

2.4.1   I/O in Periodic and Explicit-time Tasks

In principle, it is OK to perform any i/o function from periodic and explicit-time tasks. It is best not to do i/o to/from the lisp listener or any window that isn't exposed. Use functions like `w:pop-up-prompt-and-read` and `utils:pop-up-msg` for maximum safety.[3] While a task is waiting for input, no other tasks execute.

2.4.2   I/O from CPU-time Tasks

i/o from cpu-time tasks is particularly difficult because the scheduler can't tell that the task is waiting for input. To do i/o, you should only use functions that have been properly configured with `dont-swapout-function`. Currently, it is safe to use the following functions:[4] `w:pop-up-prompt-and-read`, `tv:careful-notify`, and `tv:mouse-confirm`. When i/o is done from a cpu-time task, the cpu time accounting may become slightly inaccurate. i/o should only be done from cpu-time tasks for debugging purposes. This is because any time the i/o takes is charged to the cpu task. This isn't a problem with the other task types.[5]

2.5   Debugging

When a task gets an error, the scheduler notices the error and pops up a window that includes the task-handle, the function where the error occurred and a stack backtrace. To deactivate the task, just move the mouse off the window. The execution of all the other tasks can be continued. To enter the debugger, click on the window and an error message will appear on the screen. To select the debugger for the task, type TERM META-S. It is possible to proceed from the debugger, but it doesn't always work as expected. The best thing to do is use the debugger to find and fix the problem, and then Reset and start again.

---

[3] Currently, `*terminal-io*` (and therefore all the other stream variables) are bound in the task to a deexposed background window. An error will result if output is directed to it.

[4] see "*PH:TASKS;SCHEDULER.LISP*" for further details.

[5] INTERNAL NOTE: If the scheduler could tell if a cpu-time task is waiting for i/o, these problems would disappear.

Chapter 3

Task Reference Manual

cpu-usec/internal-time [*Variable*]

fire-system [*Variable*]

> Bound within all subprocesses to the current fire-system.

time-units-per-second [*Constant*]

> Number of internal time units per second.

1-second-compute [*Function*]

> This takes one cpu-second on an Explorer-II.

1/5-second-compute [*Function*]

> This takes 1/5 cpu-second on an Explorer-II.

base-time [*Function*]

> Base time (in seconds).

brief-time-stamp *internal-time &optional (stream nil) (current-time (current-time))* [*Function*]

> Prints only those aspects of 'internal-time' which differ from the current time. Never prints seconds.

continuation-format *format-string &rest args* [*Function*]

> Like 'label-format' except that no time or task is printed (but space is left for them). Useful for continuing 'label-format' messages.

cpu-usec->internal-time *usec* [*Function*]

cpu-usec/internal-time->minutes/cpu-sec *time* [*Function*]

current-time [*Function*]

> Elapsed time (in internal time!).

debug-format *format-string &rest args* [*Function*]

> Prints debugging messages to the Lisp Listener pane in the Phoenix system.

estimated-time [*Function*]

> Return a quick approximation of the time (in internal time units).

exact-time [*Function*]

> Return the time as exactly as it can be determined (in internal time units).

exact-time-stamp *internal-time &optional (stream nil) (base-time (base-time))* [*Function*]

Prints 'internal-time' in the format {M}M/{D}D [H}H:MM.SS.

find-task *handle*   [*Function*]

        Finds the task associated with the handle. This does no error checking.

free-operations *&body body*   [*Macro*]

        Evaluate forms, but don't charge CPU time to the process doing the evaluation.

hours->internal-time *hours*   [*Function*]

internal-time->hours *internal-time*   [*Function*]

internal-time->minutes *internal-time*   [*Function*]

internal-time->seconds *internal-time*   [*Function*]

internal-time->useconds *internal-time*   [*Function*]

kill-process *process*   [*Function*]

        Really kill a process, no matter what its state is. Unwinds are handled.

label-format *format-string &rest args*   [*Function*]

        Formats 'format-string' and args on the Pheonix message label pane and the trace
        pane.

label-format? *predicate format-string &rest args*   [*Function*]

        A conditional version of 'label-format'.

make-message   [*Function*]

message-available-at-time *message*   [*Function*]

message-channel *message*   [*Function*]

message-from *message*   [*Function*]

message-send-time *message*   [*Function*]

message-text *message*   [*Function*]

message-type *message*   [*Function*]

minutes->exact-internal-time *minutes*   [*Function*]

minutes->internal-time *minutes*   [*Function*]

minutes/cpu-sec->cpu-usec/internal-time *time*   [*Function*]

parse-to-internal-time *time-string*   [*Function*]

        Parses 'time-string' into internal-time format.

popup-stop *&optional format-string &rest args*   [*Function*]

        Stop the system immediately from within an execution method.

real-time   [*Function*]

        Current time (in seconds).

seconds->internal-time *seconds*   [*Function*]

swap-in-scheduler   [*Function*]

        Allow the scheduler to run another task.

swap-in-scheduler-if-necessary *task*   [*Function*]

        Allow the scheduler to run another task unless we are the task that is going to be
        run.

:activate   [*Method of* task]

```
:after :init &rest ignore                                    [Method of task]

:after :deactivate                                           [Method of task]

:closure                                                     [Method of task]

:cpu-time                                                    [Method of task]

:cpu-usec/internal-time                                      [Method of task]

:deactivate                                                  [Method of task]

:edit-parameters &optional additional-items                  [Method of task]
```

See (:method standard-agent :around :edit-parameters) to see how to add items.

```
:handle                              .                       [Method of task]

:initial-args                                                [Method of task]

:initial-method                                              [Method of task]

:kill                                                        [Method of task]
```

Kill a task. The task is deactivated first.

```
:name                                                       [Method of task]

:period                                                     [Method of task]

:restart-time                                               [Method of task]

:schedule-type                                              [Method of task]

:state                                                      [Method of task]

task-active-p task                                          [Function]
```

Non-nil if the task is active.

```
task-dont-swapout task                                      [Function]

task-wait wait-string interval fn &rest args                [Function]
```

Wait until a specific event occurs. The fn is tested every interval in the scheduler
process. The first argument to wait-fn is always the time at which the function is
being called. The function can return T, or the time at which the task should wake
up (maybe past or future)

```
task-wait-for-interval interval                             [Function]
```

Wait until a time interval has passed.

```
task-wait-until-time time                                   [Function]
```

Wait until a specific time.

```
task-format format-string &rest args                       [Function]
```

Like 'label-format' except that the task-name and exact time are also printed.

```
time-only-stamp internal-time &optional (stream nil) (base-time (base-time))     [Function]
            print-seconds
```

Prints 'internal-time' in the format [H]H:MM or [H]H:MM.SS if print-seconds is non-
nil.

```
time-stamp internal-time &optional (stream nil) (base-time (base-time)) print-seconds    [Function]
```

Prints 'internal-time' in the format [M]M/[D]D [H]H:MM or [M]M/[D]D [H]H:MM.SS
if print-seconds is non-nil.

```
tsend task &rest args                                       [Function]
```

Send a message to a named task.

useconds->internal-time *usec*   [*Function*]

useconds->minutes *usec*   [*Function*]

useconds->seconds *usec*   [*Function*]

Chapter 4

Phoenix Task Scheduler

The scheduler is responsible for making sure each task gets allocated the appropriate amount of cpu time. Given the notion of the scheduler queue (from chapter 2) and task-time, the basic algorithm is:

1. Current task = pop(queue)

2. Current time = task-time(task)

3. Allow task to be scheduled on the cpu

4. When either: current task is a cpu-time task and a quantum break occurs (on the Explorer this happens every second), or the task isn't a cpu-time task and the task relinquishes control (by changing its restart-time and swapping in the scheduler), disable the task from being scheduled on the cpu.

5. For cpu-time tasks increment task-time(task) by the product of the task's cpu-time-to-real-time ratio and the amount of cpu time used between steps 3 and 4.

6. For non cpu-time tasks, set task-time(task) to the the task's restart-time.

7. Insert the task back into the queue. The queue is sorted by task-time (earliest first).

8. Repeat.

When an error occurs, the scheduler automatically stops to allow debugging (see section 2.5 for details).

4.1 Scheduler Implementation

This section describes the implementation of the Phoenix task scheduler on the Explorer. The scheduler is split into two parts. The first is responsible for selecting which task to execute and enabling that task. This part is implemented as a simple-process and is responsible for overall control of the task scheduler. The second part runs every time a process associated with a task gets swapped out by the Explorer operating system. This occurs when either a timer interrupt occurs (once a second), or a process explicitly swaps itself out.

17

4.2   Top Level Scheduler Loop

This section describes the first part of the task scheduler implementation. The scheduler is an instance of the task-scheduler flavor, which provides instance variables for the scheduler queue and more. The scheduler is either running (allowing tasks to run) or stopped (not allowing tasks to run). When running, it executes tasks until a specific simulation time. The queue is a priority queue (implemented as a heap). The tasks are ordered by task-time.

The main loop is as follows:

```
;; If stopped, or currently running a task or no tasks available,
;;    do nothing.
If *current-task* is non-NIL Or we're stopped Or the queue is empty
    return

;; Find the next task ready to run.  (Since tasks can have a wait
;; function, it may not be the task at the head of the queue).
Loop
    task = head(queue)
    If task-time(task) > run-until-time then
        stopped = true
        return (from main loop)
    pop(queue)
    If task has no wait function, Or the wait function returns True,
        exit loop
    ;; If the wait function fails, reschedule the task for later
    task-time(task) = task-time(task) + task-wait-interval(task)
    insert(task, queue)
Endloop
;; At this point, the task can run.
update screen (timestamp and scheduler status-window)
wait-function(task) = NIL
*current-task* = task
start-cpu-time(task) = current-cpu-time(task)
enable(task)
```

This pseudo-code selects which task to run, then enables it.

4.3   When Processes Swap Out

The second part of the scheduler implementation disables a task's process and manages task-time accounting. The Explorer operating system has been modified to call the after-task-execution function every time a process associated with a task swaps out. The pseudo-code for this part is as follows: (NOTE: process is the process associated with *current-task*)

```
;; Allow the debugger to run.
If process is in an error state return
;; Allow i/o.
If *current-task* has disabled swapping (dont_swapout_task(task) = true)
    return

If *current-task* is not a cpu-time task
    If restart_time(task) has been set or task returned :RESTART-OK
        task_time(task) = restart_time(task)
        ;; Allow another task to be scheduled.
        insert(task, queue)
        disable(process)
        *current-task* = NIL
        return
    Else
        return (allow task to continue)

If *current-task* is a cpu-time task
    delta = current_cpu_time(task) - start_cpu_time(task)
    If suspend_cpu_accounting(task)
        Then task_time(task) = task-time(task) +
                                cpu_time_adjustment(task)
        Else task_time(task) = task_time(task) +
                                delta*cpu_ratio(task) +
                                cpu_time_adjustment(task)
    ;; A task can tell the scheduler to modify its cpu usage
    cpu_time_adjustment(task) = 0
    insert(task, queue)
    disable(process)
    *current-task* = NIL
```

If the process was not disabled, the Explorer operating system would continue to execute it. When a task should be discontinued (for now), the process should be disabled and *current-task* should be set to NIL (to allow another task to be executed).

It is difficult to tell when a process is in an error state. To do this, we put advice around the function that enters the error handler.

When a cpu-time task wants to do i/o, it cannot be disabled during the i/o wait. To allow i/o from a cpu-time task, a task is given the ability to say "don't swap me out." But, when doing i/o, the task should not be charged for the time it takes a person to enter the input. To provide for this, a task can suspend cpu time accounting.[1] See section 2.4 for more details.

4.4 Portability Issues

How difficult would it be to implement this task model on some other hardware? If each task didn't have to run in its own process, it would be easy. Unfortunately, this isn't the case because tasks (especially cpu-time tasks) can be interrupted at any time and need to save current state on the runtime stack. The fine-grained process control needed to deal with non-cpu-time tasks is easy, because those tasks *know* when they need to be swapped out. Cpu-time tasks are more difficult because they need to be disabled based on some external event (timer interrupt or the passage of a specific amount of cpu time). Since these events are operating system dependent, changes may need to be made to the operating system. I can think of one implementation that will work without

---

[1] INTERNAL NOTE: This whole thing can (and should be) cleaned up.

operating system modifications. The implementation requires several things: the OS works on a round-robin scheme, each process can enable/disable another process, and each process has access to the cpu-accounting of another process. The idea is to have a scheduler task with the following loop:

```
Loop
    enable appropriate task
    repeat
        ;; If this can't be done, try sleep(small-amount-of-time)
        Swap out to allow round-robin scheduling to occur.
        If *current-task* is cpu-time, disable it.
    until *current-task* is disabled
    update accounting
Endloop
```

This partial solution doesn't deal with issues of user interface and shared data.

4.5   Errors and Debugging

Each instance of fire-system has its own scheduler instance. Thus, it is possible to have more than one Phoenix running at a time. When the Explorer warm boots, it reinstalls the original version of the Explorer scheduler. You should not try to run Phoenix after a warm boot![2] If you hear the "bomb-drop" beep, that means that an error has occurred in the scheduler. To debug this problem, go to the 'process-display' configuration. If you don't want to debug it (more likely), just reset the system with Reset.

4.6   Programming Interface

It is possible to interact with the scheduler programatically. The function current-scheduler returns the current scheduler flavor instance. The methods that can be used are described in chapter 5.

---

[2]INTERNAL NOTE: Actually, one might have reasonable success using the install-task-scheduler function to reinstall the Phoenix scheduler after a warm boot, but it is not for the fainthearted (or those who have not saved all their editor buffers).

Chapter 5

Scheduler Reference Manual

current-scheduler *[Variable]*

        A pointer to the current instantiation of the Phoenix Task Scheduler.

current-task *[Variable]*

        A pointer to the currently executing task.

previous-task *[Variable]*

        A pointer to the last task executed.

query-task-errors *[Variable]*

        If non-NIL, errors are handled interactively. The default is T.

scheduler-error-message *[Variable]*

        Holds the error message when a task error occurs.

scheduler-error-where *[Variable]*

        Holds the location of the error when a task error occurs.

scheduler-swapin-count *[Variable]*

        A counter containing the number of times the Phoenix scheduler runs.

current-scheduler *[Function]*

        Returns the task scheduler of the current fire system.

dont-swapout-function *fn* *[Macro]*

        Adds advice to the function which prevents it (or more correctly, the task running it) from being swapped out while it is being run.

in-current-task-p *[Function]*

        Return T if currently executing in *current-task*.

:dequeue-task *task* *[Method of task-scheduler]*

:edit-parameters *[Method of task-scheduler]*

:enqueue-task *task &optional (time (current-time))* *[Method of task-scheduler]*

:kill *[Method of task-scheduler]*

        Kill the scheduler.

:macro-step-scheduler *&optional (n 1)* *[Method of task-scheduler]*

Run the system for 'n' macro steps. If 'n' is NIL, run it forever.

:reset *&rest ignore*   [*Method of* task-scheduler]

Set the scheduler to an initial state.

:run *n*   [*Method of* task-scheduler]

Run the system for 'n' minutes. If 'n' is NIL, run it forever.

:run-until-time   [*Method of* task-scheduler]

:single-step   [*Method of* task-scheduler]

Run the system for the smallest time interval possible; in other words, run the system for one internal-time unit or until a task swaps out.

:stop *&optional (wait nil)*   [*Method of* task-scheduler]

Stop the scheduler.

:trace   [*Method of* task-scheduler]

without-task-swapout *&body body*   [*Macro*]

Execute 'body' without ever swapping this task out for another task. Code that does user i/o or grabs locks should be included here.

Chapter 6

Map

Maps in Phoenix represent topographical features for a land area. The features include ground-cover, elevation, roads, rivers, buildings and fire. The map of Yellowstone is approximately 75 kilometers square. It uses different representations for the various types of information present. The rest of this chapter describes these representations with some illustrative examples.

6.1 Map Basics

6.1.1 Units and Positions

All units of distance are represented in meters. A position on the map is represented as a point data structure. A point structure contains the x and y coordinates of a point in meters. The position 0,0 is the upper left of a map. Unless otherwise specified, map access functions take positions as a point. In general, the values of the x and y coordinates should be integers in the range [0, ₁width − in − meters₊] and [0, ₁height − in − meters₁] respectively.

To create a point at position x=123 and y=45000:

```
(setf p (point 123 45000)) ==> (123 . 45000)
(point-x p) ==> 123
(point-y p) ==> 45000
```

The x and y components of a point can be accessed and set with the functions point-x and point-y.[1] The map reference chapter describes many of the point functions and geometry functions that operate on points.

6.1.2 Grid Arrays

A grid-array is an array-based representation of a map. Each array element corresponds to a square region of the map. The size that an element corresponds to is called the *grid-array-size* and the *resolution* of the grid array is the log of the size to base 2. For instance, the size 256 corresponds to the resolution 8. If the world is 76,000 meters on a side (as is our Yellowstone map), a grid-array representation of Yellowstone at resolution 8 (size=256meters) is a 300x300 array. At resolution 7,

---

[1] Points are just represented as cons cells, but that doesn't mean you should use car, cdr and cons to manipulate or create them.

the array size is 600x600. Grid-arrays are typed arrays. The term *cell* is used to denote an element of a grid-array.

6.2   The Map Representation

A map is represented as an instance of the firemap flavor. There are a large number of functions and methods that operate on firemaps. The set of functions is described in chapter 7.

6.2.1   Ground Cover

The map representation currently contains 11 types of ground cover. They are agriculture (fields), chapparal, hardwood, lake, marsh, meadow, rocky, softwood, suburban, and urban areas. The eleventh is named boundary, and is used as a sentinel around the borders of the map. Ground cover is represented as a grid array at resolution *gc-cell-resolution* (8). So each cell in the ground-cover map corresponds to an area *gc-cell-size* (256 meters) on a side. Each element of the grid-array is of type (mod 16). Given a firemap named firemap, the functions cell-ground-cover and ground-cover-name can be used to access ground cover as follows:

```
(cell-ground-cover (point 34000 12000) firemap) ==> 3
(ground-cover-name 3) ==> "Lake"
```

There are eleven constants, each corresponding to the different type of ground cover. *gc-agriculture*, *gc-boundary*, *gc-chapparal*, *gc-hardwood*, *gc-lake*, *gc-marsh*, *gc-meadow*, *gc-rocky*, *gc-softwood*, *gc-suburban* and *gc-urban*.

6.2.2   Elevation

Elevation is represented as a grid array of type (mod 65536) at resolution *elevation-cell-resolution* (8). Elevations are represented in meters above sea level. Since the elevation data is stored as a grid, it is necessary to do some form of interpolation to prevent large discontinuities. For example, suppose the part of the elevation array looks like

```
30   40   50   ...

44   50   40   ...

45   40   30   ...

 .    .    .   ...
```

Since each cell represents an area of 256 meters on a side, uninterpolated the elevation at (253 . 0) = 30, (254 . 0) = 30, (255 . 0) = 30 and (256 . 0) = 40! Notice the discontinuity at the cell boundary. Phoenix interpolates elevation as follows: Imagine the cell at (0 . 0) from above. The elevation at each corner of the cell is fixed by the elevation data.

```
30---40
 |   |
 |   |
 |   |
44---50
```

Since three points define a plane, a diagonal through the cell defines two planes:

```
30---40
 | /|
 | / |
 |/  |
44---50
```

When you ask for the elevation of a position, the elevation is interpolated. This technique guarantees that elevation is continuous over the entire map (If there are supposed to be discontinuities–cliffs–they are lost). The function cell-elevation returns the interpolated elevation (rounded to meters).

```
(cell-elevation (point 0 . 0) firemap) ==> 30
(cell-elevation (point 128 . 0) firemap) ==> 35
(cell-elevation (point 256 . 0) firemap) ==> 40
```

cell-approx-elevation returns the uninterpolated value directly from the grid array.

```
(cell-approx-elevation (point 0 . 0) firemap) ==> 30
(cell-approx-elevation (point 128 . 0) firemap) ==> 30
(cell-approx-elevation (point 256 . 0) firemap) ==> 40
```

6.2.3   Roads, Rivers and Buildings

Roads, rivers and buildings are represented as *features*. Each feature type has a specific width. Currently there are eleven types of features.

f-building Buildings. (8 meters wide)

f-fireline Fireline. (4 meters wide)

f-river128 128 meter wide river.

f-river64 64 meter wide river.

f-river32 32 meter wide river.

f-river16 16 meter wide river.

f-river8 8 meter wide river.

f-river4 4 meter wide river.

f-road16 16 meter wide road.

f-road8 8 meter wide road.

f-road4 4 meter wide road.

Each feature type has a specific width (while not a requirement, feature widths are powers of two for historical reasons). In the map, features are represented as directed line segments. The feature-edge structure contains a feature type, start point and end point. Since each feature-edge has a width (the width of the feature type), the shape of a feature edge is:

```
          -----------
              |
          S----+------E
          ----|------
```
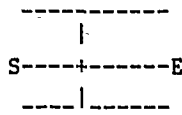
Draw a feature edge

A point is on a feature-edge iff the distance from the point to the line segment from the start point to the end point is less then 1/2 the width of the feature.

Feature types are split into two types: static and dynamic. Static features never change while the program is running. Dynamic features do change. (Currently, the only dynamic feature is fireline.) There are several reasons for this separation:

- Firemaps are large. By guaranteeing that some features are static, we can reuse the same static data structure objects in many firemaps.

- Space/Time tradeoff. The data structures used to represent static features are large in terms of space, but extremely fast to index. If each copy of the firemap was large (no data sharing), this tradeoff couldn't be made. Dynamic features are indexed more slowly, but in a more space efficient data structure.

- Cleanup. By marking some parts of the map as dynamic, it is easy to reset the map to its initial state—just clear the dynamic parts.

6.2.4   Static Features

There are two indexing methods for static features. The first method answers the question "What feature-edge is at a specific position in the map?" The second answers the question "Given a feature edge, what are the other 'adjacent feature edges?"

To facilitate the position to feature-edge mapping, a grid array at resolution *feature-cell-resolution* (8) is used. Stored in each cell is a list of all feature-edges (containing static features) that *touch* the cell in any way. A feature-edge touches a cell if any part of the rectangle defined by the segment and width cover any part of the cell.[2] The question "What feature-edge is at a specific point?" is answered by searching the list of feature-edges in the cell to see if the point is on any edge. This algorithm is quite efficient. Most cells (90 percent or more) contain no features, so no searching is done, and when a cell does contain features, there are usually only two or three. To answer the question "Is a point on a feature?" requires finding the distance from the point to the line segment of the edge. This can also be implemented very efficiently.

Since it is possible for a point to be on more than one feature, a precedence of features is defined. The order of precedence is building, roads (wide to narrow), river (wide to narrow). This way, if a road crosses a river, a point that is on the road over the river is considered to be on the road.

To answer the question "Given an edge, what edges are connected to it" several data structures are used. All edges start and end at a vertex. Each vertex contains a position and a pointer to all of the edges that start or end at that position. Since edges point to vertices and vice versa, it is possible to follow roads and rivers. For this to work properly anytime *two edges can only meet at a vertex, edges are not allowed to cross*. So when features are added to the map, the system automatically creates vertices at the appropriate places and splits single edges into multiple edges. For instance:

---

[2] So the circular positions at the edge of the cell aren't included. This isn't a problem because of the way feature-edges are connected.

```
Before:  *=vertex

     *----------*  edge1

     *----------*  edge2

After (add a vertical edge)

        * e3
        |
e6 *----*-----*  e1
        | e4
e7 *----*-----*  e2
        |
        * e5
```

Adding a single edge of feature results in two existing edges being split into four edges, and the new edge into three edges. Four new vertices are created.

6.2.5 Dynamic Features

Dynamic features are also represented with feature-edges though indexing them is different. Instead of using a grid-array to index features by position, a simple sparse array representation is used. We use a vector representation for the array.[3] To access the list of static-features in a cell at a specific point:[4]

```
(getf (svref vector (round (point-x point) *feature-cell-size*))
       (round (point-y point) *feature-cell-size*))
```

This saves significant space (a 300 element vector instead of a 300x300 array) at a cost of some speed. Since there are usually 5—10 firemaps being used at a time (in the current system), we deemed that this was an appropriate tradeoff. It works reasonably well because (in current applications) the vector is sparse.

We do not represent vertices for dynamic features. Dynamic feature may intersect at non-end points. The cost of this decision is that tracing along dynamic features is more difficult; the advantage is that the cost of creating dynamic features is low.

6.3 Fire

Fire is r presented using a grid-array at resolution 7. Thus, the grain size of a fire is 128 meters. The element type of the fire grid array can be either: bit , (mod 16)[5], fixnum[6] or T.

Currently, there are five fire states. no fire, low fire, hot fire, smoldering fire and burned out. These states correspond to different points in a fire's burn-cycle. Each state is a number, corresponding to

---

[3]INTERNAL NOTE: A hash table is another possible representation we might want to investigate.

[4]Actually, instead of round we use (lsh ... *feature-cell-resolution*). The use of power of two math saves a significant amount of cpu time.

[5]INTERNAL NOTE: Why isn't this (mod 8)?

[6]The use of fixnum instead of (mod 24) enables us to use all of the logical number operations. The drawback is that it makes the code non-portable because other Common Lisp implementations might have different length fixnums

the constants *cs-no-fire* (0), *cs-low-fire* (1), *cs-hot-fire* (2), *cs-smoldering-fire* (3) and *cs-burned-out-fire* (4).

Each of the grid-array element-types can be used to represent fire to differing levels of detail.

bit This type represents either the presence or absence of fire. The absence of fire is *cs-no-fire*: the presence of fire is *cs-low-fire*.

(mod 16) With this type each of the five fire states can be represented.

fixnum This type uses the low order bits (four bits) to represent fire state, and the high order bits to store other information. Currently, the 20 high order bits are used as flags. Functions and variables can be used to mask out the high and low order bits.

T With element type T, the data in the grid-array is either a fixnum as above, or an instance of a fire-info structure. Fire-info structures are used by the fire simulation. This structure contains a fixnum (fire-state) as above, and information used by the fire simulation such as the ignition time of cell.

When you create instances of a firemap, you can specify what element type the fire grid-array should be, or specify that a fire array shouldn't even be created.

6.4 Creating and Editing Firemaps

To create a firemap, create an instance of the firemap flavor. When a firemap is created, the new map shares information about ground cover, elevation and static features from a default firemap (*default-firemap*). The default firemap is a map of Yellowstone National Park. Firemaps can be read and written from disk, edited and examined. Care must be taken when editing maps because all maps have common data-structures. Each map gets its own set of dynamic features (initially empty) and fire representation. When instantiating a map, the type (if any) of the fire representation should be specified.

```
;; create a map of Yellowstone with a bit fire representation
(make-instance 'firemap :fire-element-type 'bit)

;; create a map of Yellowstone with no fire representation
(make-instance 'firemap :fire-element-type NIL)

;; create an empty map, and then fill if from a disk file
(setf map (make-instance 'firemap :initialize-from NIL))
(send map :load-map "map-filename")
```

The firemap creation options are described in chapter 7.

When a firemap is no longer needed, it is best to reclaim the memory used by the map (they are large) by sending it a :deallocate message. In order to return a map to its initial state (no fire, no dynamic features), send it an :erase-fire message.

The basic map access functions are:

cell-ground-cover *point map* returns the ground cover as a number (eg., the value of *gc-hardwood*).

cell-elevation *point map* returns the interpolated elevation in meters.

cell-feature *point map* returns the feature at *point* as a number (eg., the value of *f-road4*) or Nil if there is no feature.

cell-fire-state *point map* returns the fire state as a number. The type of the number depends on the fire element type. If the fire element type is T and the cell contains a fire-info structure, the fire state from the structure is returned as a fixnum.

Some access functions are implemented as functions, some as methods, and some as both. The choice of technique to use for each is based on efficiency, convenience and style.

6.5 The Real World Firemap

There is one firemap which is used to represent the "real world". When tasks look into the world, they should look into the real world firemap. The function real-world-firemap returns that map. Tasks can change the "real world." A typical change is the placement of fireline (this should be done be sending a message to fire-system rather than real-world-firemap).

6.6 Geometry

Phoenix provides many functions for geometric reasoning over the map. These functions range from simple unit conversions to basic geometry (do two line segments intersect?) to region-growing algorithms. Chapter 7 describes these functions.

Chapter 7

Map Reference Manual

7.1   Firemap Flavor

firemap-dynamic-edges *firemap* [*Function*]

:dynamic-edges [*Method of* firemap]

firemap-edge-vector *firemap* [*Function*]

:edge-vector [*Method of* firemap]

firemap-elevation *firemap* [*Function*]

:elevation [*Method of* firemap]

firemap-filename *firemap* [*Function*]

:filename [*Method of* firemap]

firemap-fire *firemap* [*Function*]

:fire [*Method of* firemap]

firemap-fire-extents *firemap* [*Function*]

:fire-extents [*Method of* firemap]

firemap-firemap-windows *firemap* [*Function*]

:firemap-windows [*Method of* firemap]

firemap-ground-cover *firemap* [*Function*]

:ground-cover [*Method of* firemap]

firemap-objects-to-display *firemap* [*Function*]

:objects-to-display [*Method of* firemap]

firemap-static-edges *firemap* [*Function*]

:static-edges [*Method of* firemap]

firemap-update-windows *firemap* [*Function*]

:update-windows [*Method of* firemap]

firemap-vertex-vector *firemap* [*Function*]

:vertex-vector [*Method of* firemap]

7.2   Map Definitions

7.2.1   Elevation

elevation-cell-resolution   [*Constant*]

elevation-cell-size   [*Constant*]
>   Size of elevation grid cell side (in meters).

7.2.2   Fire States

cs-burned-out-fire   [*Constant*]

cs-low-fire   [*Constant*]

cs-hot-fire   [*Constant*]

cs-mask   [*Constant*]

cs-no-fire   [*Constant*]

cs-smoldering-fire   [*Constant*]

fire-cell-resolution   [*Constant*]

fire-cell-size   [*Constant*]
>   Size of fire grid cell side (in meters).

fire-names   [*Variable*]

fs-features-present   [*Constant*]

fs-ignitable   [*Constant*]

fs-mask   [*Constant*]

fs-not-ignitable   [*Constant*]

number-of-fire-states   [*Constant*]

deffire *symbol number name &key color-character b&w-character color*   [*Macro*]

fire-name *fire-state*   [*Function*]
>   Returns a string which describes 'fire-state'.

7.2.3   Fire Info

allocate-fire-info   [*Function*]

fire-burn-state *f*   [*Macro*]
>   Return the burn state from a fire-info fixnum. *cs-no-fire* -> *cs-burned-out-fire*

fire-flag->number *fs*   [*Function*]

fire-flags *f*   [*Macro*]
>   Returns the flag part of a fire-info fixnum.

fire-info-burn-state *fire-info*   [*Macro*]

fire-info-change-time *fire-info*   [*Function*]

fire-info-ignite-time *fire-info*   [*Function*]

fire-info-point *fire-info*   [*Function*]

fire-info-state *fire-info*	[Function]
free-fire-info *arg*	[Function]
firep *f*	[Macro]

        Return T if there is fire in a fire-info fixnum (includes *cs-burned-out-fire*).

ignitable-p *f*	[Macro]

        Returns T if there is no fireline in a fire info cell.

live-fire-p *f*	[Macro]

        Return T if there is live fire in a fire-info fixnum (does not include *cs-burned-out-fire*).

not-ignitable-p *f*	[Macro]

        Returns T if there is fire line in a fire info cell.

7.2.4   Features

all-features-flag	[Constant]
dynamic-feature-flags	[Variable]
f-building	[Constant]
f-fireline	[Constant]
f-river128	[Constant]
f-river16	[Constant]
f-river32	[Constant]
f-river4	[Constant]
f-river64	[Constant]
f-river8	[Constant]
f-road16	[Constant]
f-road4	[Constant]
f-road8	[Constant]
feature-cell-resolution	[Constant]
feature-cell-size	[Constant]

        Size of feature grid cell side (in meters).

feature-names	[Variable]

        Array of feature names.

feature-overlay-order	[Variable]

        The precedence of features.

feature-widths	[Variable]

        Array of feature widths (in meters).

number-of-features	[Variable]
point-feature-flags	[Variable]
river-flags	[Variable]
road-flags	[Variable]

road-or-uncrossable-river-flags   [*Variable*]

static-feature-flags   [*Variable*]

uncrossable-river-flags   [*Variable*]

deffeature *symbol name number width &key roadp riverp uncrossable-river-p b&w-character*[*Macro*]
                color-character dynamicp color b&w minimum-display-size pointp

feature-name *feature*   [*Function*]

feature-width *feature*   [*Function*]

roadp *feature*   [*Function*]

riverp *feature*   [*Function*]

7.2.5   Ground Cover

gc-agriculture   [*Constant*]

gc-boundary   [*Constant*]

gc-cell-resolution   [*Constant*]

gc-cell-size   [*Constant*]
             Size of ground cover grid cell side (in meters).

gc-chapparal   [*Constant*]

gc-hardwood   [*Constant*]

gc-lake   [*Constant*]

gc-marsh   [*Constant*]

gc-meadow   [*Constant*]

gc-rocky   [*Constant*]

gc-softwood   [*Constant*]

gc-suburban   [*Constant*]

gc-urban   [*Constant*]

ground-cover-names   [*Variable*]

ground-cover-types   [*Variable*]

number-of-ground-covers   [*Variable*]

burnable-ground-cover-p *ground-cover*   [*Function*]

defground-cover *symbol number name &key type color b&w*   [*Macro*]

ground-cover-name *gc*   [*Function*]

ground-cover-type *gc*   [*Function*]

vegatation-ground-cover-p *gc*   [*Function*]

7.3   Map Access Functions and Methods

defmapfn *slot &key (slot-postfix slot) (resolution (quote* *`gc-cell-resolution*`))   [*Macro*]
           Define a set of map slot access functions at a specific resolution. Specifically, this
           defines a standard "point" reader: CELL-{SLOT} <point> <map> a "xy" reader
           CELL-{SLOT}-X-Y <x> <y> <map> and a standard "point" writer function which
           is also the setf method for the standard reader. If 'slot-postfix' is specified it is used
           in place of 'slot' to generate the function names.

7.3.1 Elevation

`cell-approx-elevation` *point map* [*Function*]

`cell-elevation` *point map* [*Function*]

> Find the real elevation of a point to the nearest meter.

`:highlight-elevation` *el &optional (range 0)* [*Method of* firemap]

`:set-cell-elevation` *point new-elevation* [*Method of* firemap]

7.3.2 Features

`cell-boundary-crossed-multiply-by-features-p` *point resolution firemap &optional* [*Function*]
    (features ˇall-features-flag ˇ)

> Return T if the cell boundary is crossed more than once by features.

`cell-dynamic-edges` *point map* [*Function*]

`cell-dynamic-feature` *point map* [*Function*]

`cell-dynamic-feature-edge` *point map* [*Function*]

> Return the line feature edge under 'point'.

`cell-dynamic-features-in-area` *point map resolution* [*Function*]

> Return the set of dynamic features at 'point' and 'resolution' as a fixnum bit-array.

`cell-feature` *point map* . [*Function*]

`:cell-feature` *point* [*Method of* firemap]

`cell-feature-edge` *point map* [*Function*]

`cell-feature-flag` *point map* [*Function*]

`cell-features-in-area` *point map resolution* [*Function*]

`cell-static-edges` *point map* [*Function*]

`cell-static-feature` *point map* [*Function*]

> Return the feature under point.

`cell-static-feature-edge` *point map* [*Function*]

> Return the line feature edge under point.

`cell-static-features-in-area` *point map resolution* [*Function*]

> Return the set of static features at 'point' and 'resolution' as a fixnum bit-array.

`:create-static-edge` *from-point to-point type &key (refresh t)* [*Method of* firemap]

> Create a static edge between the specified points. If the new edge intersects any other edges at a non-vertex, create a new vertex at that intersection point.

`:delete-edge` *edge &key (refresh t)* [*Method of* firemap]

> Delete an edge.

`:delete-edge-from-array` *edge* [*Method of* firemap]

> Delete a static edge from a firemap.

`:delete-point-edges` [*Method of* firemap]

> Delete all edges that start and end at the same vertex.

`:delete-point-feature-near-point` *point &rest ignore* [*Method of* firemap]

                                     Delete a feature near point

`dynamic-edge-exists-p` *from to firemap*   [*Function*]

`:find-edge-nearest-to-point` *point within-distance*   [*Method of* `firemap`]

          Find the edge closest to point within a certain distance.

`find-point-on-some-feature-in-cell` *cell-point resolution feature-flags map*   [*Function*]

          Pick a random point on a feature that matches feature-flags in the cell.

`:find-vertex-at-point` *point &key (create nil)*   [*Method of* `firemap`]

`:find-vertex-nearest-to-point` *point within-distance*   [*Method of* `firemap`]

          Find the vertex closest to point within a certain distance.

`fireline-in-cell-p` *point firemap resolution*   [*Function*]

`:place-dynamic-feature` *from to type*   [*Method of* `firemap`]

          Draw a dynamic feature of type from point to point.

`:place-edge-in-array` *edge*   [*Method of* `firemap`]

          Insert a static edge into the firemap.

`:place-static-edge` *from-point to-point type &key (refresh t)*   [*Method of* `firemap`]

          Create an edge of type from point to point. Vertices are created at the endpoints if
          necessary. The new edge should NOT intersect other edges (except at endpoints).

`point-on-feature-of-type-p` *point firemap feature-flags*   [*Function*]

          Return the feature if point is above a feature of a specified type.

`point-on-lake-p` *point map*   [*Function*]

`point-on-road-p` *point map*   [*Function*]

`river-in-cell-p` *point firemap resolution*   [*Function*]

`road-in-cell-p` *point firemap resolution*   [*Function*]

7.3.3   Ground Cover

`cell-ground-cover` *point map*   [*Function*]

`:cell-ground-cover` *point*   [*Method of* `firemap`]

`:set-cell-ground-cover` *point cover*   [*Method of* `firemap`]

7.3.4   Fire

`cell-fire` *point map*   [*Function*]

`cell-fire-burn-state` *point map*   [*Function*]

          Return the burn state at a point in a map.

`:cell-fire-burn-state` *point*   [*Method of* `firemap`]

`:cell-fire-display-info` *point*   [*Method of* `firemap`]

          Return three values; state, change-time, ignite-time.

`cell-fire-flags` *point map*   [*Function*]

          Return the fire flags for a cell.

`cell-fire-info-structure` *point map*   [*Function*]

        Create a fire-info structure, or return the existing one for point in map.

`cell-fire-state` *point map*   [*Function*]

        Return the full state and flags at a point in the map.

`cell-ignite-time` *point*   [*Function*]

        Return the ignite time of a cell. Return most-positive-fixnum if not known.

`:erase-fire` *&optional confirm (refresh t)*   [*Method of* `firemap`]

        Reset the map to a nice, clean, no-fire state. This really should be called :reset-all-
dynamic-edges.

`:ignite-cell` *&rest args*   [*Method of* `firemap`]

`:update-cell-fire` *point new-state &optional old-state*   [*Method of* `firemap`]

`:set-cell-fire-burn-state` *point &optional (new-state `cs-low-fire`)* [*Method of* `firemap`]
        (refresh t)

`:set-cell-fire-state` *point &optional (new-state `cs-low-fire`) (refresh t)* [*Method of* `firemap`]

7.4   Other Firemap Variables and Routines

`*default-firemap*`   [*Variable*]

        Default-firemap for ground cover and elevation.

`*default-map-file*`   [*Variable*]

`*height-in-meters*`   [*Constant*]

`*width-in-meters*`   [*Constant*]

`all-firemaps`   [*Function*]

`:check-for-non-vertex-intersections`   [*Method of* `firemap`]

        See if there are any interesections that don't meet at a vertex.

`:deallocate`   [*Method of* `firemap`]

        Return the grid arrays.

`:delete-window` *window*   [*Method of* `firemap`]

`:draw-objects`   [*Method of* `f. remap`]

`:load-map` *&optional file confirm*   [*Method of* `firemap`]

`real-world-firemap`   [*Function*]

        Returns the real-world-firemap of the current fire system.

`:rebuild-vertex-and-edge-vectors`   [*Method of* `firemap`]

        Given a firemap static-edge-vector, construct a new vertex vector and clean up the
edge vector. Vertices and vertex numbers are recomputed from scratch. Everything
is recomputed from just the edge-type, start-point and end-point.

`:refresh`   [*Method of* `firemap`]

`:save-map` *&optional file confirm*   [*Method of* `firemap`]

`:validate-vertex-and-edge-vectors`   [*Method of* `firemap`]

        Do some consistency checking over features.

7.5   Grids

grid-array-aref *array x y resolution*   [*Function*]

      Use 'x' and 'y' as indices into a grid array. Can be used with 'setf'.

grid-array-ref *array point resolution*   [*Function*]

      Use 'point' as an index into a grid array at the specified resolution. Can be used with 'setf'.

set-grid-array-aref *array x y resolution value*   [*Function*]

      Use 'x' and 'y' as indices to set an element in a grid array.

set-grid-array-ref *array point resolution value*   [*Function*]

      Use 'point' as an index to set an element in grid array at the specified resolution.

truncate-to-grid *n*   [*Macro*]

7.6   Vertices and Feature Edges

copy-feature-edge *object*   [*Function*]

edges-meet-at-vertex-p *e1 e2*   [*Function*]

      Return true if the edges share a vertex.

edge-cell-list *edge resolution*   [*Function*]

      Return a list of the cells touched by edge. Each cell is returned only once. The order is random.

edges-in-area *edges point resolution*   [*Function*]

feature-edge-bot-point-from *feature-edge*   [*Function*]

feature-edge-bot-point-to *feature-edge*   [*Function*]

feature-edge-cen-point-from *feature-edge*   [*Function*]

feature-edge-cen-point-to *feature-edge*   [*Function*]

feature-edge-from-point *feature-edge*   [*Function*]

feature-edge-from-vertex *feature-edge*   [*Function*]

feature-edge-index *feature-edge*   [*Function*]

feature-edge-length *feature-edge*   [*Function*]

feature-edge-p *object*   [*Function*]

feature-edge-plist *feature-edge*   [*Function*]

feature-edge-to-point *feature-edge*   [*Function*]

feature-edge-to-vertex *feature-edge*   [*Function*]

feature-edge-top-point-from *feature-edge*   [*Function*]

feature-edge-top-point-to *feature-edge*   [*Function*]

feature-edge-type *feature-edge*   [*Function*]

feature-of-type-in-area-p *point firemap feature-flags resolution*   [*Function*]

find-point-on-edge-in-cell *edge cell-point resolution*   [*Function*]

> Given an edge, return some point on that edge within a cell. If the edge doesn't enter the cell, return NIL.

point-distance-from-edge-squared *point edge*   [*Function*]

> Return the square of the distance from the edge centerline to point.

point-on-edge-p *point edge*   [*Function*]

> Return T if point is on edge.

type-of-connection-between-vertices *v1 v2*   [*Function*]

vertex-edges *vertex*   [*Function*]

vertex-index *vertex*   [*Function*]

vertex-p *object*   [*Function*]

vertex-point *vertex*   [*Function*]

7.7 Conversion Functions

feet-per-km   [*Constant*]

km-per-mile   [*Constant*]

meters-per-chain   [*Constant*]

chains/hour->meters/internal-time *n*   [*Function*]

chains/hour->meters/minute *n*   [*Function*]

chains/hour->meters/second *n*   [*Function*]

chains->km *chains*   [*Function*]

chains->meters *chains*   [*Function*]

degrees->radians *degrees*   [*Function*]

km->feet *km*   [*Function*]

km->miles *km*   [*Function*]

km/hour->meters/internal-time *n*   [*Function*]

km/hour->meters/second *km*   [*Function*]

miles->km *miles*   [*Function*]

7.8 Geometry Functions

essentially-zero-threshold   [*Variable*]

> To test if a point is on a line, we find its distance from the line and if it's essentially zero [it might be non-zero due to round-off errors], we deem it to be on the line. This number sets the threshold for being essentially zero.

angle-between-points *p1 p2*   [*Function*]

> Return the angle of the vector from p1 to p2 (in radians).

area-of-triangle *p0 p1 p2*   [*Function*]

> Return the area of the triangle p0 p1 p2.

average-point *new-point &rest points*   [*Function*]

Computes a new [rounded] point which is the average of some points.

cell-center-point *cell-point resolution*   [*Function*]

> Return the center point of a cell at resolution.

center-point *point*   [*Function*]

copy-extent *object*   [*Function*]

create-extent *&key upper-left lower-right*   [*Function*]

direction *vector*   [*Function*]

exact-point-separation *p1 p2*   [*Function*]

> Return the distance between two points as a floating-point number. Since points range from [0,0] to [8000,8000], the result is accurate to within +/- 1 meter.

extend-segment *p0 p1 extend-amount &optional (extend-to (point))*   [*Function*]

> Extend the segment from 'p0' to 'p1' by 'extend-amount'. Returns the new segment end [p1] rounded to meters.

extent-intersection *ul1 lr1 ul2 lr2*   [*Function*]

> Given two extents, return their intersection.

extent-lower-right *extent*   [*Function*]

extent-p *object*   [*Function*]

extent-upper-left *extent*   [*Function*]

half-line-intersects-segment-p *l0 l1 s0 s1*   [*Function*]

> Returns T iff the half-infinite line [aka the ray] starting at L0 and going through L1 intersects the segment [S0,S1].

magnitude *vector*   [*Function*]

make-extent *point*   [*Function*]

make-vector *direction magnitude*   [*Function*]

> Create a vector. 'Magnitude' and 'direction' should be regular floats.

nearest-point-on-segment *pointj line-ptl line-ptk*   [*Function*]

> Find the point on the segment defined by 'line-ptL' 'line-ptK' nearest to 'pointJ'. The returned point may be 'eq' to a segment end point.

point *&optional (x nil x-specified) y (setpoint nil setpoint-specified)*   [*Function*]

> Make a new point using 'x' and 'y'. If 'setpoint' is specified it is destructively modified and returned.

point-at-resolution *point resolution*   [*Function*]

> Make a copy of a point at a specific resolution.

point-at-resolution* *point resolution*   [*Function*]

point-at-resolution-with-offsets *p1 p2 resolution*   [*Function*]

> Return a point in the same cell as p1 with the same offsets as p2.

point-difference *p1 p2*   [*Function*]

> Return p1 - p2.

point-difference* *p1 p2*   [*Function*]

> Return p1 = p1 - p2.

point-distance-from-extents *test-point upper-left lower-right &optional (solid? t)*   [*Function*]

> This returns the distance between a point and a rectangle specified by 'upper-left' and 'lower-right'. The rectangle may be solid; if it is and the point lies inside it, the returned distance is negative and is the distance from the point to the edge of the rectangle.

point-distance-from-segment-squared *pointj line-ptl line-ptk*   [*Function*]

> Copied from Bowyer & Woodark, pg. 47.

point-distance-from-square *test-point square-origin size &optional (solid? t)*   [*Function*]

> This returns the distance between a point and a rectangle specified as an origin and a size. The rectangle may be solid; if it is and the point lies inside it, the returned distance is negative and is the distance from the point to the edge of the rectangle.

point-distance-squared-from-star-polygon *point star-point polyli..e*   [*Function*]

> Return 0.0 if the point is in the polygon, or the distance squared from the point to the polygon.

point-extent-sector-code *point extent*   [*Function*]

point-in-bounds-p *point*   [*Function*]

point-in-extent-p *point extent*   [*Function*]

> Return T if point is within the extent.

point-in-star-polygon-p *point star-point polyline*   [*Function*]

point-left-of-line-p *point p0 p1*   [*Function*]

> Determine if 'point' is in left half-plane defined by the directed line segment.

point-on-line-p *point p0 p1*   [*Function*]

> Determine if 'point' lies along line defined by segment.

point-on-segment->parameter *p0 p1 point &optional error-check?*   [*Function*]

> If POINT is on the segment [P0,P1], returns the parametric specification of POINT. Actually, this works for any point on the half-infinite line [ray] from P0 through P1 to infinity. If POINT is on the line but not on the ray, the parameter returned is the negative of the correct answer; that is, the correct answer is negative, but the answer returned is the absolute value of the correct answer.

point-right-of-line-p *point p0 p1*   [*Function*]

> Determine if 'point' is in right half-plane defined by the directed line segment.

point-rotate-around-point *theta rotate-point around-point*   [*Function*]

point-rotate-origin *theta pt*   [*Function*]

point-rotate-origin* *theta pt*   [*Function*]

point round *point*   [*Function*]

> Rounds POINT. Returns a new point.

point-round* *point*   [*Function*]

> Rounds POINT. Destructively modifies POINT.

point-sector-code *point top-left bottom-right*   [*Function*]

point-separation *p1 p2*   [*Function*]

> Return the distance between the two points +/- 1 meter. Use exact-point-separation if necessary.

point-separation-and-sin-cos *p1 p2*   [*Function*]

Find the distance between points. Return the sin and cos of the angle between them.

point-separation-lessp *p1 p2 distance*   [*Function*]

Return T if distance between p1&p2 < distance.

point-separation-squared *p1 p2*   [*Function*]

Return the square of the distance between the two points. The result is a single-precision floating point number.

point-sum *p1 p2*   [*Function*]

Return P1 + P2.

point-sum* *p1 p2*   [*Function*]

Return P1 = P1 + P2.

point-wrt-line *point p0 p1*   [*Function*]

Return 0 if 'point' is on the line, < 0 if left, > 0 if right.

point-x *point*   [*Function*]

point-y *point*   [*Function*]

point= *p1 p2*   [*Function*]

pointp *point*   [*Function*]

points-in-same-cell-p *p1 p2 resolution*   [*Function*]

Return T/NIL if the two points share the same cell at the given resolution.

polyline->segments *polyline &key (closedp t)*   [*Function*]

Converts a polyline into a list of segments [represented as a list of two points].

polyline-intersects-cell-p *polyline point resolution*   [*Function*]

Returns T iff some segment of POLYLINE goes through the cell of size RESOLUTION containing POINT.

polyline-length *polyline &key (closedp t)*   [*Function*]

quick-segment-intersects-cell-p *p0 p1 cell-point resolution*   [*Function*]

True iff the segment [P0,P1] passes through the cell containing CELL-POINT at RESOLUTION. Returns the point [rounded] where the segment intersects a diagonal of the cell. [BUG: fails if the segment ends inside the cell without intersecting a diagonal.]

radians->degrees *rad*   [*Function*]

rounded-point-p *point*   [*Function*]

same-side-p *line-pt1 line-pt2 point1 point2*   [*Function*]

Return T if 'point1' and 'point2' are both in the same half plane determined by 'linep1' and 'linep2'.

segment&parameter->point *p0 p1 parameter*   [*Function*]

Given a segment and a parameter, return a point on the segment. The point is not rounded.

segment&parameter->point* *p0 p1 parameter*   [*Function*]

Given a segment and a parameter, return a point on the segment. The new point is not rounded and is returned in P0.

segment&parameter->point1* *p0 p1 parameter*   [*Function*]

> Given a segment and a parameter, return a point on the segment. The point is not rounded and is returned in p1.

segment&parameter->point2 *p0 p1 parameter p2*   [*Function*]

> Given a segment and a parameter, return a point on the segment. The point is rounded and is returned in p2.

segment-cell-intersection *p0 p1 cell-point resolution*   [*Function*]

> Return the first intersection point of the segment [P0,P1] with the cell containing CELL-POINT at RESOLUTION. If there is no intersection, returns NIL. The returned point isn't rounded. Caution: The border of a cell is defined by four segments, the North, South, East and West edges. Points on the North and West edges are *in* the cell, while points on the South and East edges are *not* in the cell; they are in the cells to the South and East, because those same segments are the borders of other cells, and each point can only be in one cell. Essentially, the points *in* a cell are (x,y), such that CELL-SIZE = 2^RESOLUTION FLOOR[x/CELL-SIZE] <= x < FLOOR[x/CELL-SIZE]+CELL-SIZE and similarly for y. Note the <= versus the <. See SEGMENT-CELL-INTERIOR-INTERSECTION for an alternative function.

segment-cell-intersection-parameter *p0 p1 cell-point resolution*   [*Function*]

> Return the parameter corresponding to the first intersection point of the segment with the border of the cell. This point is not guaranteed to be in the cell, since the North and West borders of a cell are in the cell, while the South and East borders are not—they belong to the cells to the South and East, respectively. See SEGMENT-CELL-INTERIOR-INTERSECTION-PARAMETER for an alternative function. In more detail: The border of a cell is defined by four segments, the North, South, East and West edges. Points on the North and West edges are *in* the cell, while points on the South and East edges are *not* in the cell; they are in the cells to the South and East, because those same segments are the borders of other cells, and each point can only be in one cell. Essentially, the points *in* a cell are (x,y), such that CELL-SIZE = 2^RESOLUTION FLOOR[x/CELL-SIZE] <= x < FLOOR[x/CELL-SIZE]+CELL-SIZE and similarly for y. Note the <= versus the <.

segment-cell-interior-intersection-parameter *p0 p1 cell-point resolution*   [*Function*]

> Return the parameter corresponding to the first intersection point of the segment with the points *in* the cell. This point *is* guaranteed to be in the cell. See SEGMENT-CELL-INTERSECTION-PARAMETER for an alternative function.

segment-edge-intersection-parameter *p0 p1 edge*   [*Function*]

segment-feature-border-intersection-in-cell *p0 p1 cell-point resolution featureflags* [*Function*]
map

> Return the first intersection point of the segment with the border of an edge of type 'featureflags'. The returned point is not rounded.

segment-feature-centerline-intersection-in-cell *p0 p1 cell-point resolution* [*Function*]
featureflags map

> Return the intersection point of the segment with the centerline of an edge of type 'featureflags'.

segment-intersection *pk pl pm pn*   [*Function*]

> Return the point of intersection or nil. Copied from Bowyer & Woodwark.

segment-intersection-parameter *pk pl pm pn*   [*Function*]

Return the parameter of intersection on the segment [Pk,Pl]. If the segments don't intersect, return NIL. If the segments are coincident, an endpoint of [Pm,Pn] that is on [Pk,Pl] is returned.

`segment-intersection-parameters` *pk pl pm pn*   [*Function*]

Return the parameters of the intersection of the lines determined by segments [Pk,Pl] and [Pm,Pn]. The first value is the parameter on [Pk,Pl], the second on [Pm,Pn]. If the segments intersect, both parameters will be between 0 and 1, inclusive. If the lines do not intersect, NIL is returned. If the lines are coincident, the correct result is calculated.

`segment-intersects-cell-p` *p0 p1 point resolution*   [*Function*]

Returns T iff the segment [P0,P1] goes through the cell of size RESOLUTION containing POINT.

`segment-intersects-edge-type-in-cell-p` *start finish cell-point flags map*   .   [*Function*]

Return T iff the segment from start to finish intersects an edge with flags within cell-point.

`segment-side-segments` *p1 p2 width &optional (s10 (point)) (s11 (point)) (s20 (point))* [*Function*]
(s21 (point))

Return the two segments on the boundary of the argument segment (rounded).

`segment-to-implicit-line` *pk pl*   [*Function*]

Given a line segment, return the implict form Ax + By + C = 0.

`segment-to-parametric-line` *p0 p1*   [*Function*]

Given a segment, return the parameters of the line x = Xo + Fs y = Yo + Gs.

`segment-touches-edge-p` *p0 p1 edge*   [*Function*]

Return T if the segment from p0 to p1 comes in contact with edge.

`segments->polylines` *segments*   [*Function*]

Converts an unsorted list of SEGMENTS, each represented as a list of two points, into a list of polylines by joining segments.

`segments-intersect-p` *pk pl pm pn*   [*Function*]

Return the point of intersection or nil. Copied from Bowyer & Woodwark.

`set-point-x` *point value*   [*Function*]

`set-point-y` *point value*   [*Function*]

`vector-end-point` *start-point angle radius &optional (end-point (point 0 0))*   [*Function*]

Given a vector specified by 'start-point', 'angle' and 'radius', calculate and return an endpoint.

`xy-segment-to-implicit-line` *xk yk xl yl*   [*Function*]

Given a line segment, return the implict form Ax + By + C = 0.

`xy-segments-intersect-p` *xk yk xl yl xm ym xn yn*   [*Function*]

Return the point of intersection or nil. Copied from Bowyer & Woodwark.

7.9 Iteration Constructs

circle-neighborhood-radius-index   [*Variable*]

circle-neighborhoods   [*Variable*]

fire-neighborhood   [*Variable*]

square-neighborhood   [*Variable*]

do-feature-edges *(edge cell-point map &key feature-flags (edge-type :both)) &body body*   [*Macro*]

> Iterate over all edges in a cell matching feature flags. edge-type is either :both (default), :static or :dynamic. Body is repeated for both types of edges if necessary. Dynamic features are done first. A (return) will get you out of the current edge type loop, not the do-feature-edges loop!

do-point-set *(point seed-point &key (resolution (quote `gc-cell-resolution`)) (x (gensym))*   [*Macro*]
(y (gensym)) visited-point-bitarray) (&rest finish-forms) &body body

> Iterate over a bunch of points. Body adds points to the set of points. do-point-set guarantees that each point is visited only once. Use the macro (visit-xy x y) or (visit-point point) to add a point to the set to visit. Body is executed until the set of points is empty. Point can be modified by body, but its value is changed each time through the loop (so use copy-point to keep the point around). Visited points are remembered at the specifed resolution. This also defines the macro (do-point-neighbors (connectedness) &body body). Point is set to each neighbor (4 or 8 connected) in turn and body is executed. The expansion is of the form (setf point n1) body (setf point n2) body ...

do-polyline *(p0 p1 polyline &key (closedp t) (return nil)) &body body*   [*Macro*]

> Iterate over each segment in a polyline. If body doesn't do an explicit return, 'return' is returned.

do-vertex-neighbors *(neighbor connection vertex &key (edge (gensym))) &body body*   [*Macro*]

> Iterate over the neighbors of 'vertex'. 'Vertex' and 'neighbor' are structures.

dofiremap *(point &key upper-left lower-right (resolution *gc-cell-resolution*)) &body body*   [*Macro*]

> Iterate over a rectangular area of a map inclusive of upper-left and lower-right points. If the bounds aren't specified, use the full extent of the map.

doneighbors *(neighbor point &key (neighborhood *square-neighborhood*) neighborlist*   [*Macro*]
*min-radius max-radius distance angle path (resolution *gc-cell-resolution`)*
(grid-size nil) (check-bounds nil) (index nil)) &body body

> Iterate over neighbor points in neighborhood array. If 'neighborlist' is not specified, iterate over all neighbors. Note: 'neighbor' is destructively modified.

map-fire-region *function fire map*   [*Function*]

> Iterate over every point contained in the region of fire.

map-fire-to-boundary *function fire map &key (resolution `fire-cell-resolution`)*   [*Function*]
(stop-at-natural-boundaries t) (boundary-in-region-p nil)
(natural-boundaries-in-region-p nil) (visited-point-bitarray nil)

> Iterate over every point contained in the region of fire, up to the fire boundary. Stop spreading at natural boundaries.

map-pixels-on-line *function p1 p2 resolution &rest other-args*   [*Function*]

> Map over all pixels between two points. The traversal is from p1 to p2. The point argument to the function is destructively modified. If the function returns two values,

and the first value is :return, then the mapping is terminated and the second value is returned; otherwise, returns NIL.

some-pixel-on-line *predicate start finish resolution*   [*Macro*]

> Execute predicate for each point on line. If predicate returns non-nil, return that value.

visit-neighbors *connectivity*   [*Macro*]

> Visit all neighbors 4 or 8 connected from the current point. Can only be used within the lexical scope of 'do-point-set'.

Chapter 8

Interface

This chapter describes how to *interface* with Phoenix. There are three broad interfacing activities: Defining new commands, displaying things on the firemap windows and adding new types of windows on the *desktop*.

When Phoenix is started, an instance of the fire-system flavor is created. The entire user interface is based on *UCL* (TI's Universal Command Loop), therefore the fire-system flavor is built up from UCL flavors and mixins. Almost all interface commands are messages to the current fire system.

8.1 Adding New Commands

Several UCL command tables are used by Phoenix. To define a new UCL command decide which command table to add the command to or, if necessary, create a new command table. The convention used in Phoenix is that a directory of code may contain two files: *COMMAND-METHODS.LISP* and *MAKE-COMMANDS.LISP*. The command-method file contains all of the methods used to implement the commands and the make-command file contains all of the calls to define the commands and build the command tables. All command methods must be methods of the fire-system flavor. Look at some of the existing files to see how command tables and commands are defined.

8.2 I/O and Firemaps

A library of functions can be used to perform graphical operations on windows that display firemaps (eg. highlight-line). In addition, the mouse can be used to select positions and icons from a map. These functions are described in chapter 9.

8.3 Icons

An icon is an graphical object that can be displayed and moved around on a firemap. All firemaps contain sets of icons that are automatically displayed. Icons can be created, deleted and moved on a firemap. All agents in Phoenix are displayed by icons (bulldozers, watchtowers, etc.) Icons and the routines for manipulating them are described in more detail in chapter 9 .

8.4   Defining New Desktop Windows

To build a new window suitable for use on the desktop, add the utils::desktop-mixin flavor to the component flavors of the window. Write a "make" function to make a new instance of your window and initialize it – the function select-or-create-window-on-desktop should be useful for this. The define-desktop-window macro will associate the window type with the make function and will also add the window type to the list of window types that can be created by left-clicking on the desktop. The following code was written for the system to take an existing flavor, firemap-window, and build a new flavor that can be used on the desktop.

```
(pushnew 'desktop-firemap-window *standalone-flavors*)

(defflavor desktop-firemap-window
           ()
           (utils::desktop-mixin
            firemap-window)

  (:default-init-plist
    :font-map *fire-system-font-map*
    :foreground-color *fire-system-fg-color*
    :background-color *fire-system-bg-color*
    :border-color *fire-system-border-color*
    :scroll-bar-mode :maximum))

(defmethod (desktop-firemap-window :after :init) (ignore)
  (declare (ignore ignore))
  (send self :set-firemap (real-world-firemap)))

(define-desktop-window "Firemap" make-desktop-firemap-window)

(defun make-desktop-firemap-window ()
  "Create a firemap window on the desktop."
  (let* ((window (select-or-create-window-on-desktop 'desktop-firemap-window)))
    (send window :set-label '(:string
                              ,(format nil "Real World Firemap") :centered))
    (send window :expose)
    window))
```

Notice that this did some of initializations in make-desktop-firemap-window and used an :after :init method to do the rest. This is purely a matter of style.

Chapter 9

Interface Reference Manual

b&w-desktop-color	[Variable]
banded-color-map	[Variable]

A pointer to the phoenix color map.

bitmap-pathname-defaults	[Variable]

Place where bitmaps are written and can always be read.

blip-alist	[Variable]

The alist of mouse character blips that the fire-system window handles.

color->highlight-b&w	[Variable]
color->highlight-color	[Variable]
color-desktop-color	[Variable]
command-command-table	[Variable]
command-tables	[Variable]

A list of all command table names.

debug-screen	[Variable]
default-elevation-gradient	[Variable]
default-map-editor-state	[Variable]
distance-for-sensitivity	[Variable]

The pixel distance at which the mouse is considered to be on top of an object.

highlight-b&w-mappings	[Variable]
highlight-colors	[Variable]
highlight-cyan	[Variable]
highlight-orange	[Variable]
highlight-purple	[Variable]
highlight-red	[Variable]
highlight-white	[Variable]
highlight-yellow	[Variable]
initial-resolution	[Variable]

The resolution at which a firemap-window initially displays the firemap.

initial-screen-configuration   '   [*Variable*]

The initial fire-system window configuration.

leave-ghost-objects   [*Variable*]

A weird drawing variable that results in objects leaving a trail of where they have been.

phoenix-command-menu   [*Variable*]

query-about-selecting-phoenix   [*Variable*]

If true, ask for confirmation before bringing up Phoenix.

remember-highlights   [*Variable*]

If non-nil then highlights are stored so that the won't be lost during a refresh.

task-command-table   [*Variable*]

task-inspector-menu   [*Variable*]

task-menu   [*Variable*]

use-cached-maps   [*Variable*]

Determines whether the firemap bitmap caching is enabled.

use-color-p   [*Variable*]

Set to NIL to force B&W.

create-bitmaps *&optional (`fire-system` `fire-system`)*   [*Function*]

define-desktop-window *name function*   [*Macro*]

Add this window type to the menu of windows that can be created on the desktop.

display-phoenix-icon-window *&optional label &key (font `icon-window-label-font`) fg*   [*Function*]
(bg (aref `phoenix-icon` 0 0))

Display the Phoenix startup icon.

do-exposed-map-windows *(mapwindow &optional window map) &body body*   [*Macro*]

Execute 'body' with 'mapwindow' bound to each of the exposed firemap windows in succession.

domapwindows *(window &optional point (can-be-deexposed nil)) &body body*   [*Macro*]

Iterate over each of the windows viewing this map. If point is specified, point must be visible. If can-be-deexposed is nil, then the window must be exposed.

find-phoenix-system   [*Function*]

Called when SYSTEM S is hit.

:activate-all-tasks   [*Method of* fire-system]

:activate-task *&optional task*   [*Method of* fire-system]

:active-agents   [*Method of* fire-system]

:all-firemaps   [*Method of* fire-system]

:all-inferiors   [*Method of* fire-system]

:all-tasks   [*Method of* fire-system]

:base-time   [*Method of* fire-system]

:clear-highlights   [*Method of* fire-system]

```
:clear-trace-window                                     [Method of fire-system]
:deactivate-task &optional task                         [Method of fire-system]
:delete-task task                                       [Method of fire-system]
:describe-task &optional task                           [Method of fire-system]
:edit-environment                                       [Method of fire-system]
:edit-fire                                              [Method of fire-system]
:edit-task &optional task                               [Method of fire-system]
:elapsed-time                                           [Method of fire-system]
:erase-fire &optional (confirm t) (refresh t)          [Method of fire-system]
:exposed-firemap-windows                                [Method of fire-system]
:get-environment                                        [Method of fire-system]
:get-environment-parameter param &optional default     [Method of fire-system]
:inspect-task &optional task                            [Method of fire-system]
:macro-step-scheduler &optional (n 1)                   [Method of fire-system]
```
> Macro step the system n 'times'.

```
:macro-step-scheduler-and-wait &optional (n 1)         [Method of fire-system]
:meter-task &optional task                              [Method of fire-system]
```
> Turn on task metering.

```
:real-world-firemap                                    [Method of fire-system]
:refresh-all-windows                                    [Method of fire-system]
:refresh-objects                                        [Method of fire-system]
:reinitialize &optional (query t) (create-agents t)    [Method of fire-system]
:reset-and-activate-all-tasks &optional (query t)      [Method of fire-system]
```
> Reset all the tasks and the scheduler. Erase the fire.

```
:run &optional time-to-run                              [Method of fire-system]
```
> Run the system. Optionally, run it for 'time-to-run' minutes.

```
:scheduler                                             [Method of fire-system]
:select-configuration                                  [Method of fire-system]
:set-all-tasks .newvalue.                              [Method of fire-system]
:set-base-time .newvalue.                              [Method of fire-system]
:set-environment plist                                 [Method of fire-system]
:set-environment-parameter param value                 [Method of fire-system]
:set-real-world-firemap .newvalue.                     [Method of fire-system]
:single-macro-step-scheduler                           [Method of fire-system]
```
> Macro step the system one time.

```
:single-step                                           [Method of fire-system]
```
> Run the system for the smallest time interval possible.

```
:start                                                 [Method of fire-system]
```
> Start the system.

:start-fire *&optional radius point &optional point-y*   [*Method of* fire-system]

:stop   [*Method of* fire-system]

> Stop the system.

:task-menu-items *&optional (task-type t) initial-task-list*   [*Method of* fire-system]

:toggle-firemap *&optional (nextp t)*   [*Method of* fire-system]

> Make the left firemap pane in a 'two-view' configuration display a new firemap. The firemap chosen is the next one in the firemap list. If nextp is NIL, use the previous map on the list.

:view-firemap *&optional task*   [*Method of* fire-system]

fire-system-all-tasks *fire-system*   [*Function*]

fire-system-base-time *fire-system*   [*Function*]

fire-system-elapsed-time *fire-system*   [*Function*]

fire-system-real-world-firemap *fire-system*   [*Function*]

fire-system-scheduler *fire-system*   [*Function*]

:edit-parameters   [*Method of* firemap-window]

:find-object-near-point *point*   [*Method of* firemap-window]

> Returns the object nearest to point.

:firemap   [*Method of* firemap-window]

:mouse-select-window-point *&optional (mouse-doc "select a screen*   [*Method of* firemap-window]
>   *point.") &key (mouse-char*
>   *tv:mouse-glyph-hollow-circle-pointer)*

> Select a point from the map using the mouse. Return the point selected and the button clicked. Only single clicks are accepted. Mouse-M is bound to Abort (that is, it returns NIL as the point selected).

:move-object *object oid-position new-character*   [*Method of* firemap-window]

> Draw object on screen.

:object-menu *x y*   [*Method of* firemap-window]

> Called when right click near object

:origin-x   [*Method of* firemap-window]

:origin-y   [*Method of* firemap-window]

:point-on-screen-p *point*   [*Method of* firemap-window]

:point-screen-x *point*   [*Method of* firemap-window]

:point-screen-y *point*   [*Method of* firemap-window]

:redraw-cell *point*   [*Method of* firemap-window]

> Redraw a single cell.

:refresh-highlights   [*Method of* firemap-window]

:refresh-objects   [*Method of* firemap-window]

:set-firemap *map &optional (refresh t)*   [*Method of* firemap-window]

highlight-cell *point &key resolution color type alu map window*   [*Function*]

> Highlight a cell. 'type' is :filled or :outline. All visible maps are highlighted unless 'map' or 'window' is specified.

highlight-edge *edge &key full width color type alu units arrowp map window* [*Function*]

> Highlight an edge. 'width' is width of line (in units :pixels (default) or :meters). 'type' is :solid or :dashed. All visible maps are highlighted unless map or window is specified.

highlight-extent *p1 p2 &key width color type alu map window* [*Function*]

> Highlight a rectangular region. 'P1' can be either a point or an extent structure. 'type' is :filled, :outline, or :dashed. All visible maps are highlighted unless 'map' or 'window' is specified.

highlight-fire *fire &rest args &key (spokes nil) (color 'highlight-cyan') (projections* [*Function*] *nil) &allow-other-keys*

highlight-fire-cell *point &key color type alu map window* [*Function*]

> Highlight a cell at fire resolution. 'type' is either :filled, :outline. All visible maps are highlighted unless 'map' or 'window' is specified.

highlight-line *from-point to-point &key width color type alu units arrowp arrowheadsize* [*Function*] *fill-arrow-p band clip map window*

> Highlight a line between two points. 'width' is width of line in 'units' (:pixels (default) or :meters)). 'type' is :solid or :dashed. If 'arrowp' is T, draw an arrow head at 'to-point'. All visible maps are highlighted unless map or window is specified.

highlight-point *point &key radius color type alu units map window* [*Function*]

> Highlight a point (draw a circle around it). 'units' is the units of 'radius' (:pixels (default) or :meters.) 'type' is either :outline or :filled (default). All visible maps are highlighted unless 'map' or 'window' is specified.

highlight-polyline *point-list &key width color type alu units arrowp closedp center map* [*Function*] *window*

> Highlight a chain of line segments defined by the list of points. 'width' is width of line (in units :pixels (default) or :meters). 'type' is :solid or :dashed. If 'arrowp' is T, draw an arrow head at the end point. if 'closedp' is T (default), the points form a closed polygon. All visible maps are highlighted unless 'map' or 'window' is specified.

highlight-vector *point angle radius &key width color type alu units arrow map window* [*Function*]

> Highlight a vector. 'width' is width of line (in units :pixels (default) or :meters). 'type' is :solid or :dashed. If 'arrowp' is T, draw an arrow head at the end point. 'length' is the length of the vector in :length-units (:meters (default) or :pixels) All visible maps are highlighted unless map or window is specified.

:add-firemap *map*   [*Method of* icon]

:after *:init &rest ignore*   [*Method of* icon]

:after-draw-character-function   [*Method of* icon]

:b&w-character   [*Method of* icon]

:before *:kill &rest ignore*   [*Method of* icon]

:bg-color   [*Method of* icon]

:color-character   [*Method of* icon]

:current-character   [*Method of* icon]

:fg-color   [*Method of* icon]

:firemaps   [*Method of* icon]

:minimum-display-size   [*Method of* icon]

:name	[*Method of* icon]
:object-size	[*Method of* icon]
:orientations	[*Method of* icon]
:position	[*Method of* icon]
:print-self *stream &rest ignore*	[*Method of* icon]
:remove-firemap *map*	[*Method of* icon]
:set-after-draw-character-function *.newvalue.*	[*Method of* icon]
:set-b&w-character *.newvalue.*	[*Method of* icon]
:set-bg-color *.newvalue.*	[*Method of* icon]
:set-color *color*	[*Method of* icon]
:set-color-character *.newvalue.*	[*Method of* icon]
:set-current-character *.newvalue.*	[*Method of* icon]
:set-fg-color *.newvalue.*	[*Method of* icon]
:set-firemaps *.newvalue.*	[*Method of* icon]
:set-minimum-display-size *.newvalue.*	[*Method of* icon]
:set-name *.newvalue.*	[*Method of* icon]
:set-object-size *.newvalue.*	[*Method of* icon]
:set-orientations *.newvalue.*	[*Method of* icon]
:set-position *.newvalue.*	[*Method of* icon]
:wrapper :set-position *&body body*	[*Method of* icon]
icon-after-draw-character-function *icon*	[*Function*]
icon-b&w-character *icon*	[*Function*]
icon-bg-color *icon*	[*Function*]
icon-color-character *icon*	[*Function*]
icon-current-character *icon*	[*Function*]
icon-fg-color *icon*	[*Function*]
icon-firemaps *icon*	[*Function*]
icon-minimum-display-size *icon*	[*Function*]
icon-name *icon*	[*Function*]
icon-object-size *icon*	[*Function*]
icon-orientations *icon*	[*Function*]
icon-position *icon*	[*Function*]
load-bitmaps *&key (background t)*	[*Function*]

Load the bitmaps for firemap bitmap caching.

make-desktop-firemap-window	[*Function*]

Create a firemap window on the desktop.

move-an-object *object &optional old-position new-character*	[*Function*]
move-rectangle-within-window *window x y width height*	[*Function*]
outline-cell *point &rest args*	[*Function*]

outline-fire-cell *point &rest args*   [*Function*]

random-highlight   [*Function*]

>   Picks a random color from ''highlight-colors''.

save-bitmaps   [*Function*]

>   Save the bitmaps for firemap bitmap caching. See *bitmap-pathname-defaults*.

save-color-map *&optional (filename "ph:fonts;banded-color-map")*   [*Function*]

>   Use this function to write the color map to disk.

set-phoenix-icon-background-color *color*   [*Function*]

spum   [*Function*]

>   S elect P oint U nder M ouse - selects a point with the mouse and returns it.

using-band *(band) &body body*   [*Macro*]

>   Use a band just in a single window. This alters the plane-mask of the window so that only the specified color planes will be changed by drawing routines:

with-highlight *(highlight-form) &body body*   [*Macro*]

>   Executes body with highlight specified in ' highlight-form' turned on then turns it off.

with-method-clipping *&body body*   [*Macro*]

>   Should be used if drawing methods are used.

Chapter 10

Fire Simulation

The fire simulation is controlled by a periodic-task which is an instance of the `fire-simulation` flavor. By default, the fire-simulation updates the fire every five minutes. The computation of rate of spread is based on ground-cover, elevation gradient, wind speed, wind direction, humidity and temperature. The simulator implements most parts of the fire model in [4], though it doesn't include season, cloud cover, time of day and slope orientation (ie. south side of a mountain) as provided in the model. Information about burn times and spotting was empirically derived.

The basic implementation idea is as follows: whenever a cell catches on fire, the ignite time of all its neighbors is computed. If a neighbor already has an ignite time, and the ignite time from the newly ignited cell is earlier, the earlier time is used. The simulator maintains a large queue of cells to be ignited, sorted by ignite time. Since fire changes state, (low->hot->smoldering->burned-out), burning cells have state change times. Burning cells are also kept in the queue.

```
While the next event in the queue occurs before the end of the
       current five minute cycle
   If the event is an ignition event
       ignite the cell
       compute its change state time
       enqueue the change event
       for all neighbors of the cell
         if neighbor is ignitable
          it = time it takes for fire to spread from cell to neighbor +
              ignite_time(cell)
          if ignite_time of neighbor is nil or it < ignite_time(neighbor)
             ignite_time(neighbor) = it
             enqueue(neighbor)
   If event is a change state event
       set cell fire state
       if cell is still burning
          compute new change state time
          enqueue(cell)
```

The "neighbor" set of a cell consists of all adjacent cells (8 connected) plus all cells a knight's move away; thus a cell has 16 neighbors. The reason for the including the knight's tour is to smooth out the shape of a fire. With only 8 neighbors, fires tend to be oddly shaped unless the wind comes from one of the eight compass points. Even with 16 neighbors the fire looks odd when the wind comes from certain directions. This can be fixed, at a cost in cpu time, by increasing the number of directions in which a fire could spread.

Before deciding to compute the ignition of a neighbor, the simulation checks to see if the neighbor is ignitable. A neighbor is ignitable if:

- The neighbor contains burnable ground cover (ie. no lake) AND

- Either there are no obstructions between the cell and neighbor (roads & rivers) or the obstruction can be jumped (tested probabilistically)

When the simulator changes a fire state it modifies the `real-world-firemap`. The simulator stores simulation information in the real-world map in `fire-info` data structures.[1]

---

[1] INTERNAL NOTE: This should be changed. The real-world map should just contain fire state, type (mod 4) . The simulation can use its own map of type T, to store the fire-info. That would really clean up the fire access code and hide the simulation's implementation from other tasks

Chapter 11

Fire Simulation Reference Manual

default-fire-increment   [*Variable*]

        Default period for the fire simulation task.

default-spotting-scale-factor   [*Variable*]

        Amount to modify the base probabilities of fire spotting. 0 = no spotting. This is a multiplier for the values in '*jump-probabilities*'.

features-that-stop-fire   [*Variable*]

gc-transition-times   [*Variable*]

        AN array containing the amount of time to change from state to state (nothing -> low -> hot -> smouldering -> burnedout) indexed by ground cover.

humidity-temperature->fuel-moisture   [*Variable*]

initial-environment   [*Variable*]

jump-probabilities   [*Variable*]

        The probability that a fire-cell containing a feature of the specified type will catch fire.

primary-wind-direction-change-interval   [*Variable*]

primary-wind-speed-change-interval   [*Variable*]

burnable-cell? *cell-point map*   [*Function*]

        T iff the cell has burnable ground cover [that is, it isn't water].

burnable-ground-cover-p *ground-cover*   [*Function*]

calculate-ros *from-gc angle slope-percent &optional (verbose nil) (randomize t)*   [*Function*]

        Given a cell on fire, calculate the rate of spread in the direction of to-cell. Return a value in meters/minute.

cell-stops-fire-p *p-map &optional (resolution *gc-cell-resolution*)* [*Function*]

chop-up-segment-at-nonburnable-cells *p0 p1 map*   [*Function*]

        Returns a new list of segments on the segment [P0,P1] which don't cross any nonburnable cells [lakes], but with endpoints that are in the lakes.

copy-fuel-model *object*   [*Function*]

create-fire-info   [*Function*]

current-wind-direction   [Function]

fire-simulation   [Function]

:highlight-queued-for-ignition   [Method of fire-simulation]

:ignite-cell *point &rest ignore*   [Method of fire-simulation]

> Interactively set cell on fire.

:recalculate-all-ignite-times   [Method of fire-simulation]

:recalculate-ignite-time *point*   [Method of fire-simulation]

> Recalculate the ignite time for the specified point. This should be called anytime something happens that may effect the ignite-time (ie., fire-line, retardent, wind change, etc.).

:set-update-maps *.newvalue.*   [Method of fire-simulation]

fuel-model-moisture->base-ros *fuel-model*   [Function]

fuel-model-non-array-wind-factor *fuel-model*   [Function]

fuel-model-p *object*   [Function]

fuel-model-slope-percent->slope-factor *fuel-model*   [Function]

fuel-model-type *fuel-model*   [Function]

fuel-model-wind->wind-factor *fuel-model*   [Function]

pfi *&optional (point (spum))*   [Function]

> P rint F ire I nfo at 'point' to *standard-output*. Used for debugging.

randomize-wind-magnitude *mag*   [Function]

> Modify wind mag. +/- 10%.

rate-of-spread *gc direction grade*   [Function]

re-calc-ignite-time *point &optional (combination-method :minimum)*   [Function]

> Recalculate the ignite time for the specified point. 'combination-method' is either :set or :minimum.

ros *p1 p2 map*   [Function]

> Return the rate of spread of the fire from p1 to p2 in meters/second. Return NIL if fire can't spread over the ground cover.

simulate-forward *&optional (steps 1) (refresh t)*   [Function]

> Run the fire simulator for 'steps' steps of its default period. If 'refresh' is non-nil the display is updated.

time-to-ignite *gc distance angle delta-elevation*   [Function]

> Return the time ignite-cell should catch on fire (in seconds)

wind-magnitude-fn *magnitude angle*   [Function]

Chapter 12

Miscellaneous Functions Reference Manual

firemap-area   [*Variable*]

         GC area where firemap data is stored.

fpi   [*Constant*]

         pi as a single float.

assocf *alist item &optional default*   [*Function*]

assocq *alist item*   [*Function*]

break-string *string length*   [*Function*]

         Break a string at spaces and hyphens across several lines of length.

call-stack *&optional (process current-process)*   [*Function*]

         Return a list of the functions on the call stack.

deletef *item list &rest delete-args*   [*Macro*]

         Same as (setf list (delete item list delete-args)).

div2 *i &optional (power 1)*   [*Macro*]

         Divide positive fixnum i by 2 or a power of 2.

exp2 *n*   [*Macro*]

         2^n

flag->object *flag*   [*Macro*]

log2 *n*   [*Macro*]

         Log of n to base 2.

make-initialized-array *&rest inits*   [*Function*]

max-using-zero-if-nil *a b*   [*Function*]

mod2 *n power*   [*Macro*]

         Find n mod a power of 2.

name-of *object*   [*Function*]

         Tries to return a reasonable handle for object.

nice-call-stack   [*Function*]

        Return a call stack starting at the right spot.

nmerge-list *list1 list2 compare-fn &key (key (function identity))*   [*Function*]

        Destructive merge of list2 into list1. Both lists are may be changed. This works best
        if the lists are NOT cdr-coded.

object->flag *object*   [*Macro*]

object-in-resource-p *resource-name object*   [*Function*]

        Determine if object is in the free pool of resource RESOURCE-NAME.

  ject-set *&rest objects*   [*Macro*]

        Forms a set out of a bunch of objects. Same as (set-add (object->flag o1) (object-
        >flag o2) ...).

ordered-insert *elt list &optional (test-fn (function <)) key*   [*Function*]

        Insert elt into an ordered set.

ordered-insertf *item list &rest args*   [*Macro*]

        Same as (setf list (ordered-insert item list args).

ph-apropos *string &key predicate boundp fboundp*   [*Function*]

        Looks for STRING in the Phoenix package only.

ph-who-calls *symbol-or-symbols*   [*Function*]

        Print who calls 'symbol-or-symbols' in the Phoenix package.

phoenix-float *n*   [*Macro*]

        A faster version of (float n 1.0f0).

remassoc *alist key*   [*Function*]

removef *item list &rest delete-args*   [*Macro*]

        Same as (setf list (remove item list delete-args)).

set-add *set &rest flags*   [*Macro*]

set-addf *set &rest flags*   [*Macro*]

set-assocf *alist key value*   [*Macro*]

        Alist must be a setffable form. Shouldn't be a function returning an alist.

set-clear *set &rest flags*   [*Macro*]

set-clearf *set &rest flags*   [*Macro*]

set-test *set flag*   [*Macro*]

some* *(element list) &body body*   [*Macro*]

        An iterative version of the function 'some'. Bind each element of list in turn. If body
        returns non-nil, return the result. This is faster than the regular function, especially
        when lexical variables are used in the predicate.

square *x*   [*Function*]

times2 *i &optional power*   [*Macro*]

        Multiply by a power of 2.

trunc-coord *coord*   [*Macro*]

trunc-x *point*   [*Macro*]

trunc-y *point* [Macro]

trunc2 *n power* [Macro]

        Truncate n to a power of 2.

truncate-to-factor *n factor* [Macro]

with-phoenix-package *&body body* [Macro]

Appendix A

File Organization

The Phoenix sources are distributed among various directories on the logical host "*PH:*".

"*AM;*" Definitions of the standard agent model, communication, reflexes and sensors.

"*AU;*" Agent utilities.

"*BD;*" Bulldozer KBs and definitions.

"*COLOR;*" Color enhancements.

"*DFB;*" Distributed .reboss KBs and definitions.

"*DOC;*" Documentation.

"*FB;*" Fireboss KBs and definitions.

"*FC;*" Fuel carrier KBs and definitions.

"*FD;*" Fuel depot KBs and definitions.

"*FIREMAP;*" Firemaps and firemap window definitions.

"*FONTS;*" Fonts and color tables.

"*FS;*" Interface and top level definitions.

"*IN;*" Instrumentation utilities.

"*MAPS;*" Compiled map files.

"*PA;*" The Phoenix agent definitions and utilities.

"*PATCHES;*" Patch files and directories.

"*PH;*" The top level Phoenix directory.

"*PL;*" Plane KBs and definitions.

"*SCENARIOS;*" Scenarios and scripts.

"*SIM;*" Fire simulator code.

"*TASKS;*" Tasks and task scheduler code.

"*WT;*" Watchtower KBs and definitions.

Appendix B

System Loading and Maintenance Reference Manual

do-phoenix-init   *[Variable]*

        De.ermines whether the initializations on 'phoenix-initialization-list' are run when the system is loaded.

phoenix-initialization-list   *[Variable]*

        This initialization list is run during the initialization of the Phoenix system.

phoenix-package   *[Variable]*

        The "Phoenix" package.

phoenix-system-generic-files   *[Variable]*

        A list of all generic pathnames that make up phoenix. This is needed to make sure that no pathnames are GC'ed away.

standalone-flavors   *[Variable]*

        List of all flavors that should be compiled.

add-phoenix-initialization *name form*   *[Function]*

        Add a form to be run at Phoenix initialization time.

compile-phoenix-for-load-band *&optional force-increment-patch-version*   *[Function]*

        Compiles and loads Pheonix, rebuilds flavors and clears patches.

find-component-systems-with-files *system*   *[Function]*

        Return the list of component-systems for a system. The components are returned in an order suitable for compilation.

make-phoenix *&rest make-system-args*   *[Function]*

        Run make-system for all Phoenix subsystems.

make-phoenix-specific *&rest make-system-args*   *[Function]*

        Run make-system for all Phr .iix specific systems.

make-phoenix-utils *&rest make-system-args*   *[Function]*

        Run make-system for the Phoenix External Utils system.

phoenix-patch-level   *[Function]*

        Returns the number of patches that have been made to the current Phoenix system.

Glossary

band A bit pattern used to mask the color lookup table during graphics drawing operations. Also known as a *color plane mask*.

cell An element in a grid array.

command typein The process by which a user uses the keyboard and/or mouse to select a predefined interfacing action to execute.

cpu time The representation of time that an Explorer process maintains.

desktop An overlapping window interface to Phoenix.

Explorer Texas Instrument's standalone Lisp workstation.

Explorer II A high-performance Explorer based upon a TI's VLSI Lisp microprocessor.

feature edge A representation of a topographical surface feature which has a start point, an end point and a width.

firemap A comprehensive data structure which represents topographical information such as ground cover, roads, rivers, buildings, fire and firelines.

firemap pane A tiled window component of the Phoenix window interface which displays the firemap.

grid array A two dimensional matrix which holds topographical information.

internal time The representation of time which is used by the internal data structures of the testbed.

neighborhood A descriptive specification using relative positions of the cells near a cell in a grid array.

pointset A data structure consisting of a set of points. Often used in region-growing algorithms.

polyline A data structure consisting set of lines. Used to define curves.

process An Explorer process. A set of sequential operations in shared virtual address space with a program counter, stack of function calls and special-variable bindings.

simple process A process that does not save its state between calls.

simulation time The representation of time which is presented to the user by the Phoenix interface.

task The basic computational organizational unit in the testbed. Built upon Explorer processes.

task time The representation of internal time that a task maintains.

70

Bibliography

[1] Adrian Bowyer and John Woodwark. *A programmer's geometry*. Butterworths, 1983.

[2] Paul R. Cohen, Michael Greenberg, David Hart, and Adele Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, Fall 1989. also Technical Report 89-61, COINS Dept, University of Massachusetts.

[3] David M. Hart and Vipul Gupta. The phoenix user manual. Technical report, COINS Dept., University of Massachusetts, Amherst, MA, 1990.

[4] National Wildfire Coordinating Group, Boise, Idaho. *NWCG Fireline Handbook*, November 1985.

Index